修士論文

Javaバイト コード実行における
データ再利用の特性

指導教官　　富田 眞治 教授

京都大学大学院情報学研究科
修士課程通信情報システム専攻

山田 克樹

平成13年2月9日提出

Javaバイト コード実行における
データ再利用の特性

山田 克樹

内容梗概

われわれの確立した，Java仮想マシン (JVM) におけるメソッドを単位としたデータ再利用機構の詳細な分析を行い，メソッド処理数で19〜76%，実行命令数で2〜30%の省略に成功するなどの有効性を明らかにした．また，再利用に成功する入れ子構造に含むメソッド数が平均して1.03個に満たないこと，および，ネイティブメソッドが場合によって大きな障害となるといった問題点も明らかになった．現在のわれわれのデータ再利用機構では，実行履歴を無制限に再利用表（RB:Reuse Buffer）へ登録するが，その中の60〜90%は一度も再利用されずに廃棄されている．しかし，将来に再利用可能な処理をその他の処理からある程度明確に分離・特定できる見込みが明らとなった．例えば，再利用に関わるメソッドはRB登録に関わるメソッド全体の中でも半数以下と限定されており，また，その中でも集中的に再利用を受けるメソッドとそうでないメソッドに分かれている．今後，集中的に再利用を受けるメソッドとその入力データの具体的な規則性を発見するべくさらに精緻な分析を行い，将来に再利用される処理を予測する投機的な機構を基礎とした，選択的なRB登録，事前のRBエントリ作成など，格段に有為な高速化手法へとわれわれのRB方式を発展させたいと考えている．

Analysis of Data Value Reuse on Java Bytecode Execution

Katsuki YAMADA

Abstract

We analyzed our data value reuse technique for Java virtual machine (JVM) in detail.We found the dynamic number of reused method process reaches to 19%〜76%, the dynamic number of reused instructions reaches to 2%〜30%. Besides these desirable characteristics, we also found some problems such that the average number of method process reused at a time is near to 1 and the high frequency of native method call in some cases get in the way. Our system register almost all execution history to RB (Reuse Buffer) , and 60〜90% of registered data is discarded without any reuse. But, we found it is possible to detect reusable process in some extent. For example, the number of methods composing reused process is under 50% of the number of methods composing RB-registered process. Moreover, we found a few methods are related to almost all of reusable process. We will analyze the characteristics of key methods, their input data, and process of them more minutely to find some concrete rules to estimate whether process will be reused or not. The speculative technique to detect reused process in the future will realize efficient RB-registration system and pre-processing system with prediction of the input data to the process.

Javaバイトコード実行における データ再利用の特性

目次

第1章　はじめに

Java —現代のコンピュータ環境における意義—

Java言語により記述されたプログラムはJavaバイトコード（中間コード）に変換されて実行される.Javaバイトコードの動作するマシンであるJava仮想マシン(以下JVM)は，その定義（データ型，命令セット，実行仕様など）のみが決められており（文献[1]），実際にはハード・ソフトを用いた様々な形式で実現されている.「Java」という語は通常，Java言語でプログラムを記述してJVMで実行するという，いわば「Java環境」全体を指すものである.

Java言語は最も近代的なプログラミング言語の一つであると言われる．これは，Java言語の仕様設計にあたって，プログラミング言語の設計における過去の様々な蓄積が十分に考慮されているためである．Smalltalk [2] に始まり，C++ [3] で本格化したオブジェクト指向プログラミングは，現在，Javaへの移行期にある[4] と言われている.

Javaは，C++言語などのこれまでのプログラム技術の遺産を基にして，ネットワーク・コンピューティングを指向して構築したコンピュータ環境である，と言うことができるのである．中間コード方式の採用は，いずれのプラットフォームに対しても平均的に高い実行効率を可能とする記述を目指したためである．また，ポインタを無くしたことなどと併せて高い安全性[1] を備えることにより，ネットワークを通したプログラムの広範囲なやりとりに適するよう考慮されている．JVMはスタックマシンとして構成されているが，これも，1) オペランドの明示的な指定の排除によるバイトコードのサイズの低減; 2) 様々なレジスタ数を持つプロセッサ上の実行で平均的に高い実行効率を発揮する; といった，ネットワーク・コンピューティングに適した性質の実現を目的として採用されたものである.

現在，インターネットなどのネットワーク社会の進展に伴い，Javaの使用が急速に広がりつつある．光ファイバなどによるギガビットネットワークなどの基盤環境の普及によって，いっそう，中心的なコンピュータ環境の一つとなってゆくことが予測されている.

[1] JVMの実行ではコンピュータ自体の実行までをクラッシュさせるようなエラーはめったに発生させることができない.

現在のJavaにおける課題

上記なようにJavaは近年極めて注目されており，Java環境に関する研究が多くなされているが，とりわけ，その実行速度に関する問題が大きな課題となっている．実行速度の問題は，1)JVMは中間コード方式を採用している; 2)JVMのバイトコードレベルでオブジェクト指向の色彩が残っており，JVMは，実行時のクラスファイル群から，メッセージの字面（シグニチュア）をもとにメソッド検索を行う; などに起因するものである[1].

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環境に関しては，多くの研究が行われてきた[5]．1980年代には，LISPやPrologなど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．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り，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 [6] 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 [7] 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 [8]，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 [9] が挙げられる.

NTTのTAO/SILENT [10, 11] では，リエゾン，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.

DeutschおよびSchiffmanによるSmalltalk-80の実装[12] などで研究された動的コンパイラ技術は現在のJVM研究においても，sun社のHotSpotVM[13] やIBM社のJalapeño [14]，などのJVMで受け継がれている.

[1] バイトコードの実行開始時において安全性の確保のために行う「検証」作業がもたらすスタートアップ時間の遅延も問題とされている.

データ再利用技術による高速化

データ再利用技術は近時，極めてホットな話題の一つとなっている．これは，命令への入力を検索キーとする表形式データを構築・検索することより，a) 単一命令; b) 基本ブロック; c) メソッド; などを単位とする処理結果を高速に求める手法である

値の局所性 (*value locality*) の概念は，Lipasti らが 1996 年に文献 [15] で紹介している．Lipasti らは，計算機命令が動的に生成するデータは，それ以前にその命令が生成したデータと等しい可能性が高く，生成データの予測に基づく処理高速化の可能性を示している．Tyson と Austin は 1997 年の文献 [16] において，SPECint ベンチマークの 44%のロード命令，SPECfp ベンチマークの 44%のロード命令が直前の同一命令の実行と等しいデータをロードすることを明らかにした．また文献 [17] では，命令の生成するデータの約 75%が冗長なものであったという報告もある．

値の局所性の概念に基づいて，投機的なデータ予測を行うデータ投機技術や非投機的なデータキャッシングを行うデータ再利用技術の研究が進められた．データ投機に関しては効率的な予測アルゴリズムに関する研究がかなり進んでいる．履歴に基づく予測法やストライド予測法，それらのハイブリッド形の予測法 [18]，さらには文脈を基礎とした予測法 [19] などのアルゴリズムの有効性が明らかになっている．データ再利用に関しては，文献 [20] ではレジスタデータを入力・出力データとして命令レベルの再利用を検討し，複数命令系列のデータ再利用も文献 [21] などでの報告がある．最近では，特別な命令を用意して，コンパイラと連係して基本ブロックなどにデータ再利用を適用する研究が報告されている [22, 23].

バイトコードに対する高速化技法としては，命令畳み込みの適用 [24, 25]，などがなされている．しかし，バイトコードの実行に対してデータ投機やデータ再利用を適用した研究は，ほとんど報告されていない．文献 [26] では，有限個のレジスタ，有限容量のキャッシュを有する 5 段パイプライン構造を設計し，Last Value Prediction を行うデータ投機では 3.8%から 29.1%（平均 17.0%），データ再利用では 1.1%から 47.0%（平均 6.7%）のサイクル数を削減できることを報告している．

JVM はプラットフォームに対する中立・独立性のためにスタックアーキテクチャを採用している．全ての演算はスタックを経由するために，RISC プロセッサ等で今日一般的となっているような，命令レベルの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技術との適性は高くない．一方で，一般的な RISC プロセッサでは，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が，JVM では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．このため，データ再利用の機構を比較的単純に構築することができ，効果も高いのではないかと考え，本研究に着手した．

本稿では，われわれが確立した，Java 仮想マシン (JVM) における，メソッドを単位としたデータ再利用機構の詳細な分析を行い，メソッド処理数で 19〜76%，実行命令数で 2〜30%の省略に成功するなどの有効性を明らかにする．また，一方で，明らかとなっている幾つかの問題点について検討を加える．現在のわれわれのデータ再利用機構では，実行履歴を無制限に再利用表（RB:Reuse Buffer）へ登録するが，その中の 60〜90%は一度も再利用されずに廃棄されている．しかし，将来に再利用可能な処理をその他の処理からある程度明確に分離・特定できる見込みが明らとなった．例えば，再利用に関わるメソッドは RB 登録に関わるメソッド全体の中でも半数以下と限定されていることが明らかとなっている．また，その中でも集中的に再利用を受けるメソッドとそうでないメソッドに分かれている．本稿の結果から，将来に再利用される処理を予測する投機的な機構による，選択的な RB 登録，事前の RB エントリ作成などの方式が実現されると考えている．

第2章　基礎概念

2.1　Java環境

Java言語[27]により記述されたプログラムはJavaバイトコード（中間コード）に変換されて実行される．Javaバイトコードは「.class」を拡張子に持つクラスファイルとして厳密な仕様が定められている．

Java言語は最も近代的なプログラミング言語の一つであると言われる．基本的な構文形式などではC++のそれを踏襲したオブジェクト指向言語であるが，C++の欠点とされた部分は大胆に排除がなされている．

C++言語などのオブジェクト指向を受け継いだ，ネットワークコンピューティングに適した統合的な近代的計算機環境として，今日のようにJavaは普及した．インターネットなどのネットワーク社会の進展に伴い，Javaの使用がますます広がりつつある．Java技術の加速度的な高度化，および光ファイバによるギガビットネットワークなど基盤環境の進展に伴い，いっそう中心的なコンピュータ環境の一つとなってゆくことが予測されている．

2.2　Java言語

Smalltalkに始まり，C++で本格化したオブジェクト指向プログラミングは，現在，Javaへの移行期にあると言われている[4]．Java言語は，今日までのプログラミング言語技術の蓄積を最も反映した最新のプログラミング言語である，と言える．Java言語は次のような特長を持つ．

- ▽ SmalltalkやC++を起源とするオブジェクト指向の設計思想．
- ▽ シンタックスはC++言語のそれを殆ど踏襲（C++の知識のみでも，およそJavaのプログラムが書ける）．
  - ▷ ループ，条件分岐，演算子といった基本的な構文．
  - ▷ C++言語と同等の，try〜catch文による明示的な例外処理の記述方式．
- ▽ 現代的な機能の，言語レベルでのサポート．（Java以前のプログラミング言語では拡張ライブラリなどで実現されていた）．
  - ▷ マルチスレッド
  - ▷ ネットワーク
  - ▷ unicode[28]
- ▽ プログラムの分かりやすさを，言語設計の面から最大限追求している．

  - ▷ 多重承継(multiple inheritance)を排除し，interfaceを導入．
  - ▷ 分岐文(goto)を完全に排除し，ラベルによる選択break文を導入．
  - ▷ 自動記憶管理機構(GC：garbage collection)の採用により，プログラマのメモリ管理を排除[1]．
  - ▷ 宣言と実装が分割されることになるヘッダファイル等を排除．
- ▽ 実装に依存しない[2]プログラミングの実現．
  - ▷ データ型の標準化（例えば，整数型は常に32bitのデータ巾からなる）．
- ▽ 安全性（計算機クラッシュに対する耐性）への十分な配慮．
  - ▷ GCの採用によるポインタの排除．メモリリークは理論上発生しない．発生する場合はVMの実装上のバグである．
  - ▷ コンパイル時の厳格な検証(verification)．

Java言語は，C++言語などのこれまでのプログラム技術の遺産を継承・発展した，ネットワーク・コンピューティングを指向したプログラミング言語である．

2.3　Java仮想マシン

2.3.1　アーキテクチャ

Java仮想マシンは，図1のような構成をとる[1]．フレームは，メソッド[3]呼出しに対応して，Javaスタック[4]上に生成（プッシュ）され，そのメソッドの終了に伴い廃棄（ポップ）される．個々のスレッドはそれぞれ個別のJavaスタックを持ち，フレームを作成する．フレームは1) ローカル変数；2) オペランドスタック；の要素を持ち，対応するメソッド内での中間結果となるデータを保持する．ローカル変数とオペランドスタックの大きさは静的に（コンパイル時に）決定され，クラスファイル内の各メソッドの記述に含まれる．

なお，メソッドコール・リターンにおいて，これらのデータ領域は次のように使われる．メソッドコール・リターンにおける処理は次のようになる．

- メソッド呼び出し命令は，オペランドスタックのスタックトップにあるデータを，呼出し先メソッド処理の引数として用いる．新フレームをプッシュ

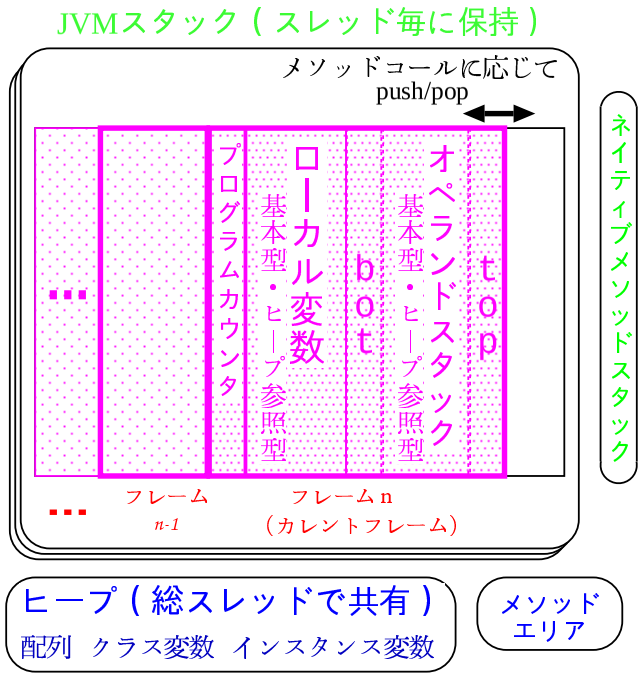
[1] メモリリークやヌルポインタのエラーはデバッグに手間がかかる．記憶管理はC++言語のプログラミングにおいて大きな作業量を要していた．

[2] Write Once, Run Anywhere.：ひとたびJava言語でプログラムを書けば，どのようなプラットフォームでも実行可能．

[3] C言語の関数にあたる．

[4] C言語などにおける「スタック」に相当する．

図 1: JVM の基本構成



し，引数をローカル変数にセットする．新フレームをカレントフレームとして，呼出し先のメソッドコードに処理を移行される．

- メソッドを終了する命令は，返り値を持つか否か，さらに，持つ場合にはその型の種類に従って 6 種類の JVM 命令が定義されている．返り値を持つ場合には，オペランドスタックのスタックトップにあるデータが返り値として用いられる．現在のカレントフレームがポップされて，必要なら，新たなカレントフレームのオペランドスタックに返り値がプッシュされて，呼出し元のメソッド処理が再開される．

ヒープは全てのスレッドで共有するデータ領域である．クラスインスタンス変数や配列はヒープに格納される．

中間コード方式の採用は，いずれのプラットフォームに対しても平均的に高

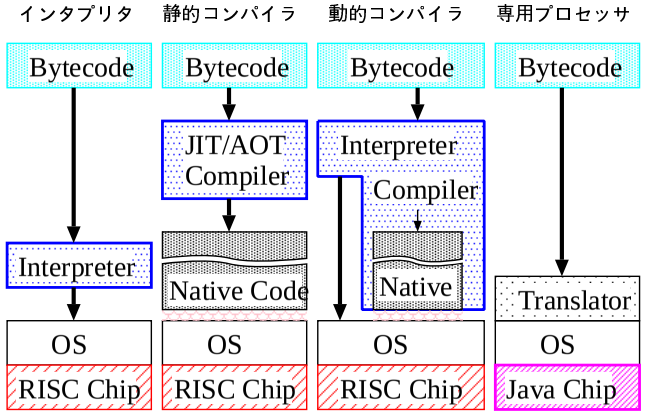
い実行効率を可能とするインプリメントを目指したためである．また，ポインタを無くしたことなどと併せて高い安全性を備えることにより，ネットワークを介した広範囲なプログラムのやりとりに適するよう考慮されている．JVM はスタックマシンとして構成されているが，これも，1) オペランドの明示的な指定の排除によるバイトコードのサイズの低減; 2) 様々なレジスタ数を持つプロセッサ上の実行で平均的に高い実行効率を発揮する; という，ネットワーク・コンピューティングに適した性質の実現が目的とされているのである．

2.3.2 実現方式

JVM の実装には，大きく以下の方式がある（図 2 参照）．

- インタプリタ：
  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．
- JIT (Just-In-Time) コンパイラ：
  クラスのロード時にクラスファイルをネイティブコードに変換するコンパイラである．この方式は，インタプリタよりも高速だが，インタプリタの数倍のメモリを必要とする．メモリ量や消費電力の制約が多い組込み用途には最適とは言えない．また，場合によっては，メモリが不足してネイティブコードに変換できなくなる問題もある．

図 2: JVM の各種実現方式の概念図



- AOT(Ahead-Of-Time) コンパイラ：
  実行時以前にクラスファイルを CPU 依存のネイティブコードに変換するタイプのコンパイラである．AOT はその性質上，本来の Java の長所であるダイナミッククラスローディングが行えないが，アドレスは静的に解決される[1]．
- 動的コンパイラ：
  最初はインタプリタまたは簡単な JIT で実行する．実行中にアプリケーションのボトルネックを検出し，このボトルネック部分に対して，強力な最適化をかけながら JIT で変換する．比較的小さいメモリで高速実行できる．
- 専用プロセッサ：
  Java の実行を前提に開発したプロセッサを用いる．ただし，バイトコードの一部[2]はソフトウェアなどで実現される．一般に，メモリ消費量が格段に少なくて済む．組み込み用途などで，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採用される．

メモリ量の制約が少ない，デスクトップ環境などでは JIT 方式が主流である．組込み分野ではコストが重視され，メモリ消費も可能な限り小さくすることが必要がある．このため，メモリ消費の大きくなる JIT コンパイラや AOT コンパイラは利用するのが難しく，専用プロセッサ方式が用いられる場合が多い．われわれは主としてハードウェア直接実行方式を念頭に置いて，データ再利用技術を JVM に適用する方式の検討を行っている．

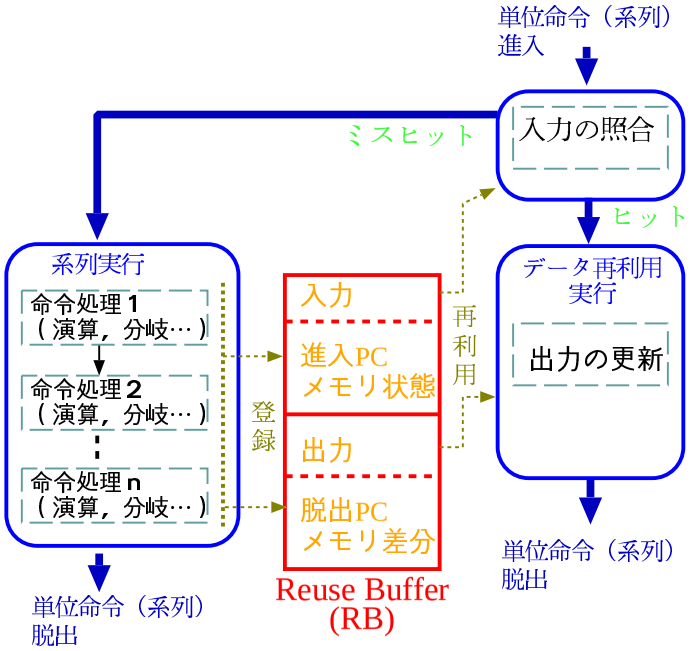
2.4 データ再利用技術

本稿では，高速化手法としてデータ再利用の手法を用いる．この手法は，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ものである．対象命令への入力を検索キーとする表形式データを構築・検索することより，1) 単一命令；2) 複数命令の系列；3) 基本ブロック；4) 基本ブロックを超える命令系列；などを単位とする処理結果を高速に求める．

データ再利用を行う場合，処理は図 3 のように進行する．再利用を行う単位命令系列への進入時に，再利用表 (RB:Reuse Buffer) 検索で，現在の PC 値とメ

[1] Hewlett-Packard 社の ChaiVM など (http://www.chai.hp.com/).
[2] GC やマルチスレッド，モニタなどの高度機能など

図 3: データ再利用の仕組み



モリ状態をキーとした比較に一致するエントリが存在しない場合には，通常の逐次実行が行われる．この時，通常の処理の他に，進入／脱出 PC 値，およびその単位系列の処理命令への入力データと出力メモリ差分情報が RB に登録される．再びその単位系列の処理を行う時に RB にヒットすると，出力メモリ差分情報により実行結果を反映するのみで，その単位系列の実行は，実際の逐次処理自体は行わずに終了する．ただし，1) 単位命令系列の中に新規のメモリを割当てる命令；2) 動的・静的な割り込み；3) 同期処理；4) 入出力；などが生じる処理の場合には特別な対応が必要となる．

2.5 評価環境

2.5.1 ベースとしたJVM

実験には現在最も有名なJVMソフトウェアの一つであるKaffe 1.0b4をベースとして用いた．FreeBSD3.2上のkaffe JVMを変更して，データ再利用技術による高速JVMを開発した．本研究のねらいはハードウェアにおける実装であるが，ソフトウェアによるインプリメントから評価・分析を行う．

2.5.2 評価用ベンチマーク

データ再利用による高速化の評価のためのベンチマークとして，SPEC JVM98 VERSION 1.03[29]を使用した．これは，compress, jess, db, javac, mpegaudio, mtrt, jackの7種類のベンチマークからなり，各ベンチマークはs1，s10，s100と3段階の実行サイズの目安が存在する[1)]．各ベンチマークにおいて行われる処理は，以下の通りである．

- compress：
  LZW法を用いてデータの圧縮を行うベンチマークである．SPEC JVM 98のcompressは，SPEC CPU 95のcompressをJava言語を用いて書き換えたものである．このプログラムは，ファイルに書かれた内容を入力として処理を行う．実行オプションの変更により，処理するデータ総量が増大される．
- jess：
  Java Expert Shell Systemの略である．十分な実行時間を得るために同じパズルを異なる文字を使用し反復して行っている．このため，実行を重ねる毎にシステムはより多くのルールを探索することとなる．実行オプションの変更は，扱うパズルの種類を変更する．
- db：
  メモリ上に存在するデータベースに対する操作を行う．ファイルに記述されたデータベースのレコードを入力として読み込む．また，データベースに対する操作もファイルに記述されており，メモリ上に読み込まれたデータベースに対してその操作を行う．実行オプションの変更により，データベースのサイズと操作数が増大される．
- javac：
  Java Development Kit(JDK) 1.0.2のJavaコンパイラである．実行オプションs1，s10，s100の指定により，コンパイルするプログラムが，hello world，jess，Lexical Analyzer Generatorと変わる．
- mpegaudio：
  MPEG 3オーディオをデコードするベンチマークである．実行オプションの変更により，デコードするオーディオのレコード時間が増大する．
- mtrt：
  レイトレーシングを行うベンチマークである[1)]．実行オプションの変更により，レイトレーシングの対象物が複雑になるとともに，画面サイズが大きくなる．
- jack：Javaプログラミング言語のパーザジェネレータによるベンチマークである．

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．また，本稿の表中において，紙面の都合上compressベンチマークは「comp」，mpegaudioベンチマークは「mpeg」とそれぞれ略記しているので留意されたい．

[1)] 本来のSPEC JVM98ベンチマークではcompressおよびmpegaudioベンチマークにおいて，処理量の倍加のために，実行オプションの値に基づく定数の回数だけ，全体をループさせることを行う．われわれの実験でこのループは，本質的な問題とはなっていなかったので取り除いた．したがって本稿で示す各ベンチマークの統計値は，ベンチマーク本来の値とは異なる．

[1)] SPEC JVM 98のmtrtベンチマークは，複数のスレッドがそれぞれレンダリングを行う．本来は，実行オプションs1においては一個のスレッドのみ用い，s10とs100においては二個のスレッドを用いる．今回のRB実装は実験的な段階であり，RBアクセスのロック処理は行わず，実行オプションs10ならびにs100においても，一個のスレッドで実行させている．

第3章　再利用を行うプログラム単位

データ再利用機構において，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，再利用を行うプログラム単位の決定は全体の性能に大きく影響する．JVMでデータ再利用を適用する単位には，a) 単一の命令；b) 単一のメソッド内の複数命令の系列；c) 複数のメソッドにわたる命令系列；などが考えられる．また，複数命令の系列を単位とする，上記のb) およびc) の場合には，1) 基本ブロック[23]；2) 一定の性質を持つJVM命令の系列；3) メソッド；などに注目して再利用単位構成アルゴリズムを決定することになる．また，単一の命令を単位とする場合には，どのJVM命令をどのような条件で実際の再利用するか，再利用し得るかを決定しなければならない．複数命令の系列を単位とする場合であっても，どのような性質を持つ命令系列が再利用可能か，再利用による高速化の効果が高いか，検討しなければならない．

3.1　個々のJVM命令の扱い

データ再利用は，全てのJVM命令に対して適用できるわけではないし，特別な機構の支援を必要とする場合や全く不可能な場合もある．また，単体では再利用が困難であるが，特定の条件下では，有効な再利用が可能なJVM命令も存在する．さらには，データ再利用技術と相性が良くて，大きなスピードアップをもたらすJVM命令と，そうでないJVM命令も存在する．本節では，JVMにデータ再利用を適用する場合における，個々のJVM命令の性質を検討する．

3.1.1　再利用が困難であるJVM命令

全てのJVM命令の中で，データ再利用が不可能ないしは困難であるものを以下に列挙する．これらのJVM命令は実行結果を再利用することが不可能であるか，または，複雑な支援機構によって実現したとしても，全体的な高速化に寄与しないことが明らかと考えられる．これらのJVM命令を含む命令系列もまた同様であり，以下のJVM命令は，データ再利用を行わないことが妥当である．

- `new` , `newarray` , `anewarray` , `multianewarray`
  これらの命令は，ヒープ上にメモリ領域を確保し，クラス・インスタンス変数または配列を生成する．入力データが同じである場合にも実行結果は異なるため，データ再利用は困難である．

- `athrow`
  例外ないしはエラーをスローする命令である．まず，現在実行中のメソッドから呼出し関係を辿ってフレームのポップを行い，例外ハンドラが存在するメソッドコールのフレームがカレントフレームとされる．そのうえで，オペランド・スタックをクリアして，例外ないしエラーを意味するオブジェクトをスタックトップへ積み，ハンドラの実行が開始される[1)]．この一連の処理は，オペランド・スタック上のオブジェクトリファレンスが一致する場合であっても結果が異なることがあり，データ再利用は困難である．
- `monitorenter` , `monitorexit`
  モニタの所有権を獲得・解放する命令である．同期のための処理を省略することは不可能であり，これらの命令またはこれらの命令を含む命令系列は原則的に再利用できない．

これらの命令は，いずれも出現頻度が十分に低いことが分かっており，データ再利用を行わないJVM命令として扱った時，全体の性能に与える影響は少ない．

3.1.2　単体での有効な再利用ができないJVM命令

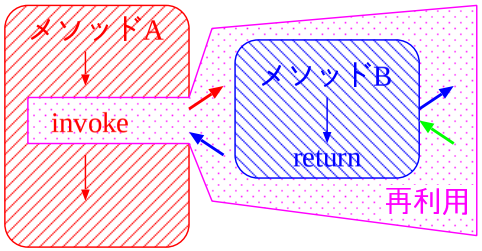
a. メソッド呼び出し・終了

メソッドを呼び出し(`invoke`)，または現在のメソッドを終了する(`return`)命令はデータ再利用でどのように扱うべきであろうか？

- `invokevirtual`, `invokespecial`, `invokestatic`, `invokeinterface`
  これらはメソッドを呼び出す命令である．引数が同じである場合にも生成するフレームは異なる．これらの命令を単独で再利用する場合，特別な機構の支援により，一定のフレームを生成するという結果自体を再利用することは可能かもしれない．しかし，RB検索によるフレーム生成は，引数から直接フレームを生成する処理よりも一般には複雑であろう．
- `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`, `return`
  これらはメソッドを終了して呼び出し元の処理に復帰する命令である．現在実行中のフレームを破棄し，元のフレームに復帰する．必要ならば帰り値の書き込みも行う．これらの命令を単独で再利用する場合，オペランド・スタックの状態だけでは復帰先のフレームを特定できず，復帰先のフレームを特定する特別な機構を用いたとしても，オーバヘッドにしかならない

[1)] 図1参照．

図 4: invoke 命令から return 命令までを単位とした再利用



と考えられる．

メソッド呼出し・終了を行う JVM 命令は，いずれも，引数または返り値となる，現在のオペランドスタック上のデータを，それぞれ，呼出し先または呼出し元のオペランドスタックに書き込む．引数や返り値をキーとして RB 検索を行い，呼出し先または呼出し元のオペランドスタックへの書き込みという処理結果を行うことは，データ再利用としての高速化の意義を持たない．結局，メソッド呼出し・終了を行う JVM 命令を，単独でデータ再利用することは，無意味である．しかし，メソッド呼び出しからリターンまでの処理をデータ再利用の単位とすれば，データ再利用を行うことが有効となる．このときは図 4 のように `invoke` 命令から，対応する `return` 命令までを単位として再利用が行われ，ネストして呼び出すメソッド処理があれば，それも併せて再利用する．

JVM のメソッドコールは，バイトコードの命令系列ではない，それぞれの環境固有のネイティブコードで記述されたネイティブメソッドを実行することがある．入出力などを扱うクラスに属するメソッドなどを中心として，ネイティブメソッドとしての記述が行われる．ネイティブメソッドに記述されるネイティブコードの処理，またはその処理を一部に含む JVM 命令の系列は，原則的に再利用できない．

b. ヒープ・ローカル変数領域，オペランドスタックのアクセス命令

ヒープに割り付けられたメモリ領域の値や，フレーム上のローカル変数領域の値を（オペランドスタックに）読み出す命令[1] において，RB に登録される

[1] この点，JVM は厳密なスタックマシンであり，ヒープ上のメモリ領域の値や，ローカル変数領域の値をオペランドスタックに読み出す命令しか用意されていない．ヒープ上のメモリ領域の値をローカル変数領域に読み出したり，ローカル変数領域の値をヒープ上のメモリ領域に読み出したりする命令はない．

「入力」と「出力」は次のようになる．

- 入力：メモリアドレスとメモリ上の値
- 出力：読み出した値のオペランド・スタックへの書き込み

すなわち，ヒープ・ローカル変数をオペランドスタックにリードする命令では，「入力」値を得れば「出力」の内容は明らかであり，RB 検索・登録処理を行うならば，それは完全なオーバヘッドに他ならない．オペランドスタックからヒープ・ローカル変数にライトする命令も同様である．

これらの命令の処理を単体で再利用して JVM の高速化を行うということは，原理的に不可能である．しかし，複数の命令を単位として実行する場合，メモリのリードは入力データの読み出しを，メモリのライトは実行結果の書き込みを意味するため，データ再利用を適用する際にはこれらの命令に注目することとなる．

また JVM にはオペランドスタックをアクセスする命令として，

- 定数のオペランドスタックへのプッシュ．
- オペランドスタックを操作（プッシュ，ポップ，複製作成など）する命令

が備えられているが，ヒープ・ローカル変数領域をアクセスする命令と同様に，単体で再利用を行うことは意味がない．

c. その他の，単体でのデータ再利用が意味を持たない命令

分岐を行う `goto` 命令など．

3.1.3 単体でも有効な再利用が可能な JVM 命令

これまでに述べた以外の命令は，データ再利用が可能な通常の JVM 命令であり，例えば，1) 演算；2) 比較；3) 型変換；4) 比較に基づく条件分岐；などの命令がこれに分類される．

3.2 単一命令を単位とする場合

Java バイトコードは基本的にオペランド・スタックのスタックトップの値を扱うため，前後の命令間でのデータの依存関係が強い．つまり，単一の命令をデータ再利用の単位とすると，ある命令がデータ再利用が可能であるか否かを判断する際に，この命令の直前の命令の実行結果を使用する場合が多い．すなわち，ある命令の実行結果が再利用できるならば，直後の命令の実行結果も再

利用できる可能性が高い．これは，われわれのグループによるいくつかの調査でも明らかとなっている [30]．

先の 3.1 節で書いたように，メソッド呼び出し・復帰命令，ヒープやローカル変数といったメモリのリード・ライト命令などを，単一の命令のみでは再利用を行うことは意味が無い．われわれは，データ再利用を適用する単位を，単一の命令ではなく複数の命令系列とする方が有効であると考えた．この点については，特別な応用などにおいては単一命令を単位とする再利用が非常に効果的であるなどの場合も考えられ，今後の検討も必要であると考えられる．

3.3 区切り命令を用いる命令系列単位

先にわれわれは，個々の JVM 命令について検討を加え，再利用を行うプログラム単位に含み得ないか，または，含まない方が妥当な JVM 命令を明らかとした（3.1 節）．原理的にデータ再利用を行い得ない JVM 命令を区切り命令として，図 5 に示すように，JVM 命令系列を複数の命令系列群に分割して，再利用を行うプログラム単位とする考え方がある．

この時，3.1 節に示したように，対応するメソッドコール・復帰命令の片方のみを含む命令系列で再利用を行うことは避けるべきであった．メソッドコール・復帰命令も区切り命令に含めるか，または，複数のメソッドにわたる命令系列でも，対応するメソッドコール・復帰命令の両方を必ず同時に含む条件を課すか，が妥当な判断である．また，区切りとなる命令の間の命令系列でも，その極大集合ではなく，何らかの基準に基く部分列で再利用を行うことも考えられる．逆に，極大集合を再利用する場合でも，含まれる JVM 命令数の個数が少ない場合には，再利用の効果が発揮されないため，排除しなければならない．

われわれは，区切り命令に分割される命令系列の極大集合で一定以上の大きさを持つものを対象としたデータ再利用について，実際の実験を行った．しかし，結果的に，次項に示すメソッドの実行を単位とするデータ再利用の方が優れている点が多いと結論付けるに到った．次項では，区切り命令を用いる命令系列単位とする方式と対比しながら，メソッドの実行を単位とする場合の説明を行う．

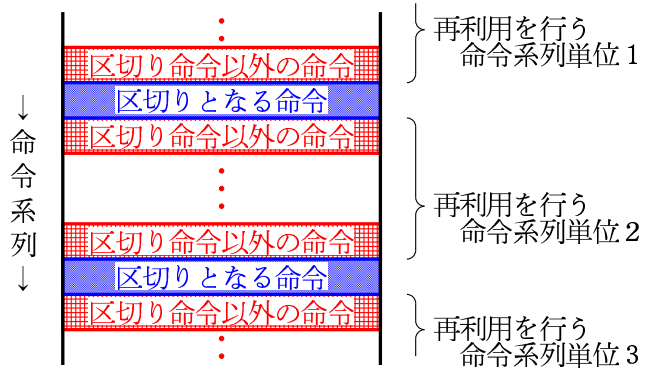


図 5: 区切り命令を用いる命令系列単位

3.4 メソッドをひとかたまりとする命令系列単位

メソッド呼び出しを行う `invoke` 命令から，その呼出しを終了する `return` 命令の間の命令系列で，再利用を行い得ない命令を含まないものを，再利用を行う単位とする方式である．

メソッド呼び出しは入れ子状に行われる．メソッドの実行を単位としてデータ再利用を行う場合，内側のメソッドおよび外側のメソッドの各々についてデータ再利用の単位とすることが可能である．例えば図 6 の場合，メソッド A の処理中でメソッド B が `invoke` され，メソッド B の内部でメソッド C がさらに `invoke` されるので，RB への登録は，メソッド A の `invoke` に始まる単位と，メソッド B の `invoke` に始まる単位で，2 エントリ行われる．内側の単位における入出力は，外側の単位における入出力ともなり，2 つのエントリに RB 登録される．また，内側の単位に再利用を行い得ない命令が存在して，再利用の対象とならないならば，外側の単位もまた再利用の対象とならない．この方式の長所は次のようなものである．

- あるフレームの生成からそのフレームの廃棄までを統合的に再利用することとなる．フレームの生成・廃棄にまつわる，煩雑な処理を，全体的に排除することができる．したがって，たとえメソッドに含まれる命令系列が帰り値の書き込みさえ行わない `return` 命令のみであったとしても，メソッドの呼び出しとリターンでそれぞれ行われる処理を省略することができ，高速化が可能である．動的なメソッド・インライン展開の効果がある [30]．

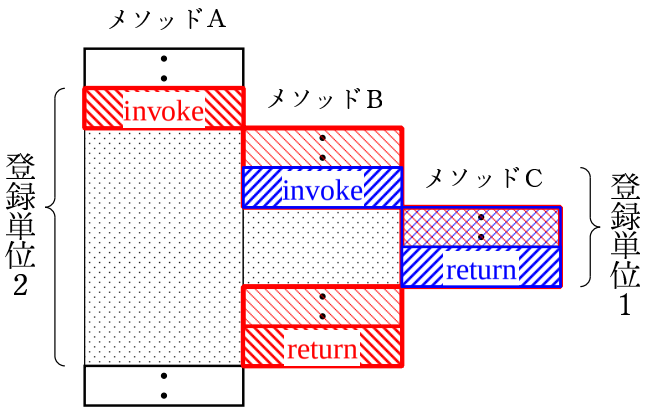


図6: メソッドの実行を再利用の単位とする場合

- またメソッドを単位として再利用を行う場合には，複数の命令系列を単位とするとはいっても，メソッドという比較的限定的な命令系列をデータ再利用の単位として扱うことになる．これによりRBの構成などははるかに簡潔となり，RB検索・登録にあたっても迅速な処理が可能となる．先に書いた区切り命令を用いる命令系列単位は，脈絡のない命令系列単位を対象としている．
- メソッドを単位とした命令系列の再利用が完了するときには，最後の`return`命令を含むメソッドに対応したフレーム，および，JVMスタック上でそのフレームの上にpushされたフレームは全て破棄されている．これらのフレーム上のデータ（オペランドスタック，ローカル変数）へのアクセスは，原則的に，RB登録の対象とする必要がない．先に書いた区切り命令を用いる命令系列単位のデータ再利用との大きな違いである．

以上のような，メソッドの実行を単位とする方式の長所より，このプログラム単位に基づくデータ再利用の比較優位性は明らかである．われわれは，メソッドの実行を単位としてデータ再利用を行う場合に注目した．次章では，メソッドの実行を単位とするデータ再利用を行う機構について検討する．

第4章　われわれの確立した方式

JVMの持つメモリ領域は，1) オペランド・スタック；2) ローカル変数；3) ヒープ；である．データ再利用が可能であるか否かの判断は，これらのメモリ領域の状態が以前実行した場合と同じであるかどうかの比較により行う．また，再利用が可能であると判断された場合には，再利用表（RB:Reuse Buffer）からこれらのメモリ領域に対して，実行結果を書き出す．

第3章での検討に基づき，われわれは，メソッドを単位とするデータ再利用方式の優位性に注目した．ここでは，メソッドを単位としてデータ再利用を行う，確立したRB機構について述べる．同時に，区切り命令を用いる命令系列単位の再利用を行う方式を採用した場合との対比も行い，われわれの方式の優位性を示す．

4.1　再利用表に保持されるデータ

RBには，再利用を行うプログラム単位そのものを特定するデータ，および，そのプログラム単位への入出力データが保持される．

4.1.1　再利用を行うプログラム単位を特定するデータ

これは，データ再利用を行うプログラム単位を指定するために保持するデータであり，データ再利用を行う単位命令系列そのものを特定する．

○ データ再利用を行う単位への進入地点

メソッドを単位とする場合には，対象となるメソッドの先頭であり，進入するメソッドの情報のみをRBに保持すれば十分である．

○ データ再利用を行う単位からの脱出先の地点

メソッドを単位とする場合，単位系列の終了地点は，進入地点の`invoke`命令に対応した`return`命令，すなわち当該`invoke`の呼び出したメソッド処理内で実行された`return`命令となる．入力の一致に基づいて，RBエントリからの出力の書き出しを行った後は，常に，当該メソッド呼び出し命令`invoke`の直後に記述された命令を実行することになり，再開開始命令アドレスをRBに明示的に保持する必要はない．区切り命令を用いてデータ再利用を行う場合，実行結果を書き戻した後に実行する命令を指定する必要がある．このため，次に実行すべき命令のアドレスもRBに保存しなければならない．

4.1.2　プログラム単位への入出力データ

データ再利用を行うプログラム単位がメモリから読み出す入力データ，および，プログラム単位の処理を終了したときに実現された出力のデータ，すなわち，メモリ状態の差分を RB に記録する．

対象となるメモリ領域は，区切り命令を用いるプログラム単位の場合は，ヒープと JVM スタック上のデータ（オペランドスタック，ローカル変数）である．一方，メソッドを単位とする場合に対象となるメモリ領域は，原則的にヒープのみであり，フレームへのアクセスは，中間結果のリード・ライトに過ぎないとみなして，無視することができる．フレームは，メソッドコールの呼出し•終了に応じて，それぞれ，プッシュ・ポップされる．再利用を行うプログラム単位がカレントフレームとしてアクセスするフレームは，全て，プログラム単位の開始後に作成されたものであり，かつ，プログラム単位の終了時には全て廃棄されている．これらのフレームへのアクセスは，単なる中間結果の記録に過ぎず，データ再利用を行うプログラム単位そのものへの入出力とはなり得ない．

ただし例外的に，次の，オペランドスタックアクセスはそれぞれ，入力，出力として，RB に登録されなければならない．

これらの処理が扱うカレントフレームは，再利用を行うプログラム単位の開始前に生成され，終了後にも維持されるものである．

○ オペランドスタックからの入力データ：
再利用を行うプログラム単位の始点となる `invoke` 命令の，呼出しメソッドに対する引数としての，そのオペランドスタックからのデータポップ．このデータは，再利用を行うプログラム単位の開始前の命令が直接に生成したデータである．

○ オペランドスタックへの出力データ：
再利用を行うプログラム単位の終点となる `return` 命令が返り値を返す場合の，呼出し元のメソッド処理のオペランドスタックへの，返り値のプッシュ．このデータは，再利用を行うプログラム単位の終了後の命令が直接に利用するデータである．

これらのデータ書き込みは中間結果ではないので，再利用を行うプログラム単位への入出力として，RB に登録しなければならない．

入出力データとして保存するデータは，入出力の値そのものと，対象となったメモリアドレスである．ただし，入出力そのものの値においては，その型に関

する情報も同時に必要である．また，ヒープ上のメモリアドレス指定は，ヒープ上のオブジェクト・クラスへの参照と，それらの中でのフィールド・スタティック指定子や配列のインデックスの組からなる．

4.2　RBの登録･検索，およびRBからメモリへの書き出しの処理

確立した RB 機構における，登録および検索の動作を以下に示す（図 7）．

1. メソッド呼び出しに対し，メソッドと引数から比較の候補とする RB のインデックスを N 個求める (a).
2. RB に登録されている過去の引数およびヒープからの読み出しデータと，現在の引数およびヒープ上データを比較する (b).
3. 再利用が可能である場合には，登録してある書き込みデータをヒープに書き出し，返り値をオペランド・スタックに取り出す．
4. 再利用が不可能である場合には，実際にメソッドの逐次実行を行い，各 RB エントリの大きさに関する制限に納まる範囲の実行結果を順次 RB に登録

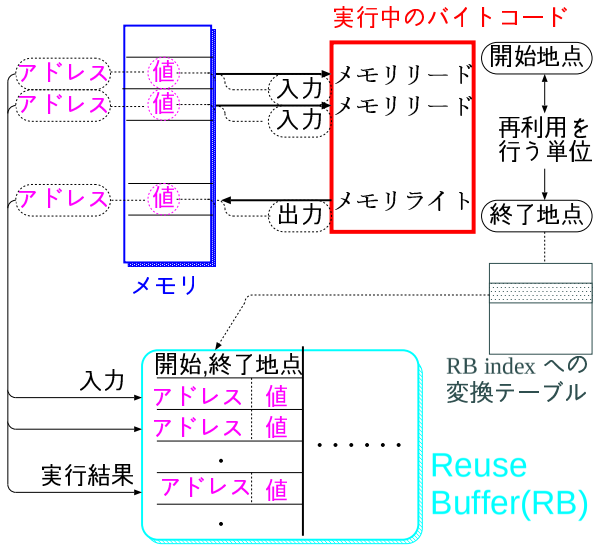


図 7: データ再利用におけるデータ流

する (c).

メソッドが入れ子になっている場合，下位のメソッドが参照したヒープに関する情報を上位のメソッドに対応するRBにも登録する．

4.3 RB登録における特別な処理

4.3.1 登録処理の無効化

以下に述べる場合には，再利用できないので，登録を無効化する必要がある．

a. ネイティブメソッドコール

ネイティブメソッドは，バイトコードではなく計算機環境特有のネイティブコードを実行するメソッドである．3.1節で書いたように，RB機構はネイティブメソッドの処理，および，ネイティブメソッドの処理を内部に含むメソッドコールの再利用を行わない．メソッドコールの時に，呼出し先メソッドがネイティブメソッドであることが分かった時点で，全ての登録処理の無効化が行われる．

b. 再利用不能な命令

再利用不能な命令が実行されると，その命令を含む全ての登録処理が無効化される．

また，割り込みの発生は，現在のメモリの状態に関わらないものである場合が多く，割り込みが発生した時点で登録中のRBのエントリはデータ再利用に使用することができず，これらのエントリは，登録を行ってはいけない．メソッドを単位としてデータ再利用を行う場合，割り込みが発生すると割り込み処理を行うメソッドの実行中に，エラーや例外などのオブジェクトの生成が行われる．このオブジェクトを生成する命令の実行により，現在登録中のRBのエントリの登録が中止される．

c. RBエントリの大きさを超える入出力

RBエントリの領域には，入力の登録がなされる部分と出力の登録がなされる部分があり，それぞれは，引数または返り値，スタティック変数，クラス変数，配列のアクセスを記録する，4部分から構成される．それぞれは，静的に大きさが定められており，登録できるデータアクセスの個数の上限が定められている．

この上限を超える入出力が発生した場合，溢れを生じる最も内側のプログラム単位，および，そのプログラム単位を含む全てのプログラム単位の登録経過

は全て無効化される．ただし，必ずしも，最も内側のプログラム単位の登録から，全ての登録処理が無効化されるわけではない．

d. 深過ぎるネスト

一定の深さ以上のネストが行われた場合には，最も深いネストを行うプログラム単位（最も外側の単位）の登録処理を無効化する．無意味に深いネストを行うプログラム単位の登録は，全体の効率に悪影響を与えると考えられるからである．

4.3.2 同一アドレスへの複数のアクセス

RBに登録を行う場合には，同一のアドレスに対して複数のアクセスが行われることがあるが，これらは，以下の条件に留意して場合分けを行って，RB登録を行わなければならない．

- プログラム単位内で生成された中間結果をリードする場合は，それは，入力データとして登録してはならない．メモリリードを行う場合に，同一アドレスへの以前のメモリライトが存在すれば，プログラム単位内での中間結果のリードに過ぎないので，登録をしてはならない．
- 中間結果としてのメモリライトは出力データとしてRB登録してはならない．メモリライトを行う場合に，同一アドレスへの以前のメモリライトは，プログラム単位内での中間結果に過ぎなくなったとして，その登録を無効化しなければならない．

すなわち，RBに登録される同一アドレスへの入力データ，出力データは，それぞれ，高々1個でなければならない．同一アドレスへの既存のRB登録の状況によって場合分けすると，表1のようになる．それぞれの状況に応じて，新規RB登録は，表1の第3列，第4列の性質をもつものとして扱わなければならない．一部のメモリリードは，中間結果のリードや，すでに登録済みの入力データのリードである．また，メモリライトは，同一アドレスへの以前のメモリライトが中間結果の書き込みに過ぎなかったことを明らかにする．

結局，メモリリード・ライトの登録は次のように行うこととなる．

- メモリライトの場合
  1. 常に新規登録を行う．
  2. ただし，同一アドレスへの既存の出力記録は上書き・抹消する．
- メモリリードの場合
  1. 同一アドレスへの既存の入出力記録があるならばRB登録しない．

表 1: 同一アドレスへのアクセスが既に RB に記録されている場合

既存の RB 登録		新規 RB 登録の扱い	
入力	出力	リード	ライト
無	無	新規の入力	出力
無	有	中間結果のリード	出力（要上書き）
有	無	登録済	出力
有	有	中間結果のリード	出力（要上書き）

2. さもなければ，このアドレスへの初めのアクセスであり，RB 登録を行う．

内側の登録と外側の登録で異なる処理が必要となる場合があるので，この場合は，それぞれに異なった対応をおこなわなければならない．

4.3.3 登録途中での再利用の実行

現に再利用単位の RB 登録を行っている最中に，先に RB 登録した再利用データの実行を行う場合には，その再利用データ全体を，現在登録中の全ての RB エントリに反映させなければならない[1)]．RB エントリ間のデータ転送が必要である．ただし，このような機構を備えなければ，それまでの全ての登録経過を無効化して現在の再利用を行うか，現在の処理に通常の逐次実行の手間をかけてそれまでの登録に順次付け加えなければならない．

4.4 RB 機構の工夫

4.4.1 スレッド間での RB の共有

JVM におけるデータ再利用機構で，われわれの方式では，複数のスレッドによる実行を行う場合においても，実行結果を登録する領域は全スレッドで共有する．各スレッドで個別に保持する方式と比べて，効率良い再利用が可能ではないかと考えたためである．しかし，各スレッドが個別に保持する方式も考えられ，個々の応用においては格別の能力を発揮する場合も考えられる．マルチスレッド処理における RB の共有・非共有の問題はこれからの検討課題の一つである．

[1)] ただし，このような事例は非常にまれにしか発生していない．

4.4.2 RB インデックスの決定

RB インデックスを求めるハッシュ関数のキーは，[メソッド：引数 1：引数 2:･･･引数 n_{arg}] とした．ここで n_{arg} は「メソッド」の引数の個数である．上キーから計算される一定個数のハッシュ値は，この処理単位の実行結果を RB に登録するインデックスの候補として定め，検索の範囲を限定する．メソッドごとに割り振られる原始的なハッシュ値は，メソッドエリア（図 1 参照）において，コード情報などに添えて格納し，管理される．この値と，引数 1〜引数 n_{arg} に基づくハッシュ関数を，RB インデックスへの変換に用いる．

ここで［メソッド］というキーを提案する向きもあるかもしれないが，RB エントリの需要は登録するメソッドの頻度と入力の多様性に依存することから，この方式は排除した．一方では，ハッシュ関数の簡単化のために，引数の数を限定または固定して，k を定数などとして［メソッド：引数 1：引数 2:･･･引数 k］をキーとする方式も考えられる．

第5章　RBの立場から見たJVMメソッド処理の分析

ここでは，JVMプログラムとその実行過程の分析を行う．われわれの確立した機構では少なくとも一つのメソッド処理を包含するプログラム処理をのデータ再利用を行う．われわれのRB機構の動作はJVMのメソッド処理の状況に関わる問題である．

5.1　使用メソッド

図8はSPEC JVM98の6種のベンチマークについて，メソッドエリア（図1参照）にロードされる．各メソッドの大きさの分布を調べたものである．ここで，メソッドの個数（Y軸）については対数表示を行い，メソッドの大きさ(Y軸) の範囲については，有為な部分のみを示した．図中では上段に中抜きの棒と，下段に黒塗りの棒の2つのデータが示してある．上段はロードされるメソッドの全体の統計である．この中で，実際の処理の対象とはならない（コールされない）ものを除いた結果が下段のデータ（黒塗りの棒）である．なお下段のデータについて，各メソッドの図中で,「all」欄の値は使用する全バイトコードメソッド数を示し,「native」欄の値はネイティブメソッドの個数を示している．

大きさが1バイトのメソッドは帰り値なしでメソッドを終了する「return」命令のみのメソッドである．例えばjessベンチマークのクラスファイル

spec.benchmarks._202jess_.NullDisplay

クラスにおける，AssertFact，RetractFactなどのメソッドは，大きさが1バイトのメソッドにコンパイルされる．

図8よりメソッドの性質に関して次のようなことがわかる．

i. コードサイズの大きなメソッドは指数関数的に減少する．
ii. 殆どのメソッドは50バイト以下のJVM命令系列で記述される．
iii. 100バイトを超えるときわめて少数のメソッドしか存在しない．
iv. 使用メソッド数は数百から千個程度である．
v. 大きさが5バイトのメソッドが最も多く，1バイトのメソッドも相当数存在する．

われわれの確立した機構では，メソッドの実行を単位として再利用を行う．メソッドの種類が，高々1200個程度あることは，再利用を行うプログラム単位を管理する機構において，十分に考慮しなければならない．

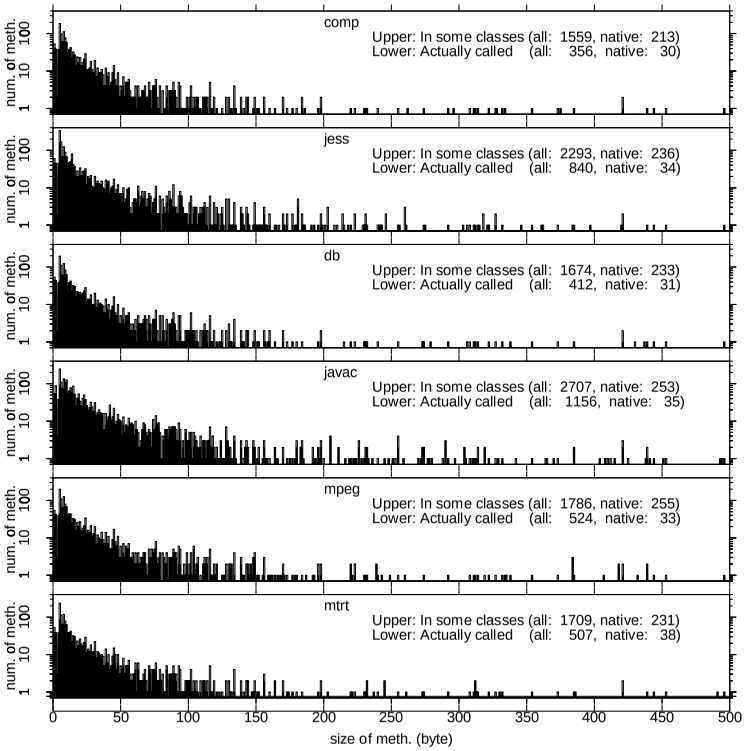


図8: 使用するメソッドの分析結果

5.2　メソッドコール

メソッドコールを行う時は，`invoke`命令が発せられて新たなフレームが作成されて，それがカレントフレームとされる．引数が，旧フレームのオペランドスタックから取り出されて，新フレームのローカル変数にセットされた後，呼び出し先メソッドのJVM命令系列の実行が開始される．呼出し`invoke`命令に対応する`return`命令，すなわち，呼び出されたメソッド内の`return`命令に到達すれば，作成したフレームが廃棄されて，返り値が存在するなら，返り値が呼出し元フレーム上のオペランドスタックにプッシュされる．

メソッドコールの発生を，メソッドエリアに記憶されるところの，JVM命令

系列の大きさに着目して分析し，実行オプション s100 の場合について図 9 にまとめた．かなり大きなメソッドも多数コールされているために，バイトコードの大きさ (X 軸) も対数表示している．図 8 の場合と同様に「all」欄と「native」欄を入れて，それぞれ総バイトコードメソッドコール数と，総ネイティブメソッドのコール数も併せて対数で示した．図 9 は実行オプション s100 の場合のデータであるが，s1，s10 の場合でも全体の分布傾向としては，有為な差は存在しなかった．

図 9 よりメソッドコールに関する次の性質がわかる．

i. コードサイズの大きなメソッドの種類は多くないが，その呼び出しは少なくはない．むしろ，メソッドコールとコードサイズの大小は無関係であり，

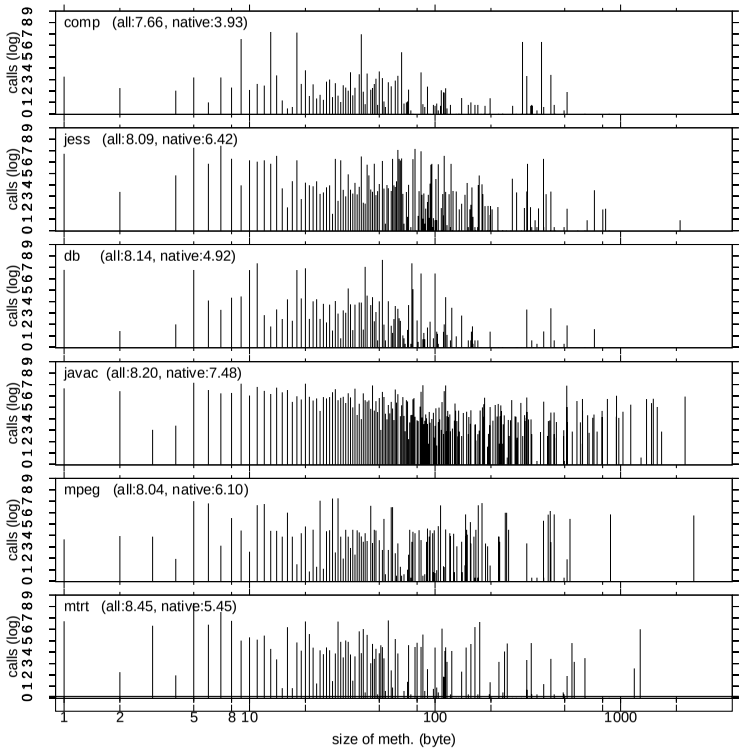


図 9: メソッドコールの分析結果 (s100)

それぞれのベンチマークの性質に左右されている傾向が強い．

ii. メソッドの処理量も考えるならば，大きなサイズのメソッドの処理は，全体の処理の相当部分を占める傾向があることが明らかである．

iii. 100 バイトを超えるメソッドは少数であることが図 8 より分かっていたが，全体の処理量の一定部分はそれらのメソッドの処理に費やされていると推定される．

iv. 少数のメソッドに関わるメソッド処理が，メソッド処理全体の大部分を占める傾向が強い．これらのメソッドに特化して対処を図ることが枢要な課題であることが分かる．

5.3 ネイティブメソッド

ネイティブメソッドを含む系列の再利用は原則的に不可能である．プログラム中のネイティブメソッドの増大は，一般に，データ再利用の効果を低減させる．ここでは，JVM におけるネイティブメソッドの位置付けについて検討することとする．ネイティブコードに関して，図 8 で示した，静的な，使用するメソッド数の統計値をまとめたので，図 10 に示す．図 9 で示した，動的なコール数の統計値をネイティブメソッドについて抽出すると，表 2（および図 11 のグラフ）のようにまとめられる．

図 10 は，使用するクラスファイル全体の中で実際にコールが行われた，バイトコードメソッドおよびネイティブコードメソッドの総数を全体と考えて，ネイティブメソッドの割合をしらべたものである．2.94% から 7.77% という割合を示している．表 2 はバイトコードメソッドコールおよびネイティブメソッドコールの総数を全体と考えて，ネイティブコードコール数の割合を調べたものである．ただし表 2 のグラフでは，各ベンチマーク欄で，3 種類の実行オプションによる結果を並べて示している．これらの図表より，次のような考察を行うことができる．

i. ネイティブメソッドの個数，およびそのコール回数が，それぞれ使用メソッド総数およびメソッドコール総数に占める割合はベンチマークとその実行オプションによって様々である．

ii. 使用メソッド数の中での割合は 2.94%〜7.77% と，各ベンチマークで，高々 3 倍程度の違いしかないが，メソッドコール数の割合に関しては 0.02%〜24.67% と，1000 倍以上異なる場合がある．相対的にばらつきのある結果が

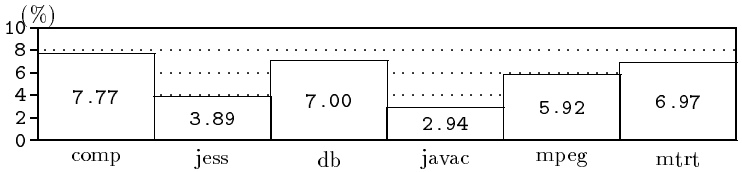


図10: ネイティブメソッドの割合

表2: ネイティブメソッドコールの割合

(%)	comp	jess	db	javac	mpeg	mtrt
s1	0.04	6.62	12.60	24.67	1.67	2.23
s10	0.04	0.87	1.99	16.44	1.73	0.75
s100	0.02	2.08	0.06	15.85	1.14	0.10

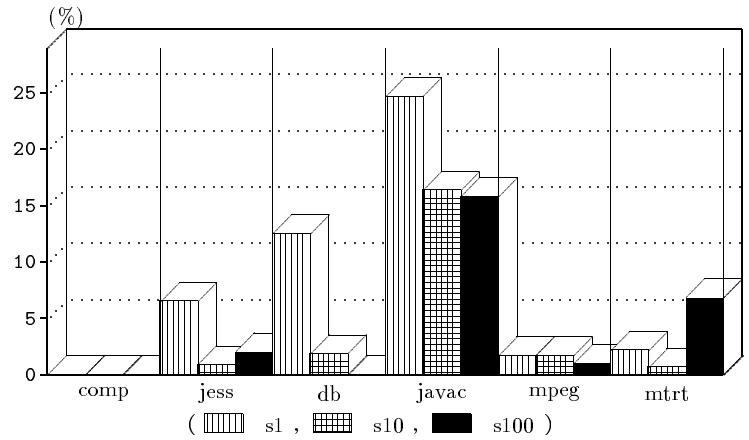


図11: ネイティブメソッドコールの割合

見られる.

iii. compress ベンチマークと mpegaudio ベンチマーク以外では，実行オプションによる変化も大きい.

実行オプションの数値は総処理量を表す目安であった．処理量の増大は概ねネイティブメソッドの比率を低下させることがわかるが，jess ベンチマークのように非線型の結果を見せるものも存在している．一部のベンチマーク実行ではネイティブメソッド処理の割合が10%を超える場合（表2の浮き出し欄）もあり，これらの場合には，RB 機構の効率に対する，相当程度の障害となることも予想される．われわれは，可能ならば，ネイティブメソッドコールを含んで再利用を行う方式も，今後の課題として考えられる.

なお，場合によってはネイティブメソッドコールの割合が相当程度大きくなることが分かったが，RB 機構によるデータ再利用を備えた場合には，RB エントリからのデータ書出しによる，バイトコードメソッドコールの省略もおこなわれるために，バイトコードの逐次実行という定常的な JVM 動作によるメソッド処理の割合は，さらに低下する.

第6章　データ再利用の分析

今回の実験で用いた，データ再利用技術を搭載した高速化JVMについて，そのRBの諸元を表3に示す．メソッド呼び出しに対するRBのインデックスは，メソッド自身と引数から比較の候補が最大512個求められる．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，返り値を保持するが，記録できるデータの上限は表3の「エントリ内の制限」に示した値である．入力データ数の上限および出力データ数の上限を超過する処理は，RB登録の対象とならない．

以下の分析では，第3章および第4章におけるRB機構の検討に従い，表3の諸元に示された規模で実現したRB機構を用いた．ただし，表3の制限はRBヒットに対して有為な差異をもたらさないことが明らかとなっているなるべく小さな値を用いた．

表3: RBの諸元

RBエントリの総数				32768
個々のメソッドが使用するエントリ数の上限				512
エントリ内の制限	入力データ数の上限	フレーム	引数	8
		ヒープ	フィールド	8
			スタティック	2
			配列	4
	出力データ数の上限	フレーム	返り値	1
		ヒープ	フィールド	4
			スタティック	4
			配列	2

ハードウェア上でのメソッド呼び出しの省略の効果を知るために，ここでメソッド呼び出しおよびリターンに要するサイクル数を仮定する．メソッド呼び出し，リターン時に行われる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する．また，ローカルレジスタのサイズは8であるとする．

このレジスタ1つにつき1サイクルで状態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必要となるサイクル数はそれぞれ18である．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する．この比較は1サイクルにつき1つのヒープ上の値に対して行うこと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．また，パイプラインハザードはメソッドの呼び出しおよびリターン時以外に一切生じないものと仮定する．これらの仮定に基づいた場合，表4に示すように，サイクル数を削減することが可能であった．

表4: ハードウェア化したRBを用いた場合の省略サイクル数(%)

(%)	comp	jess	db	javac	mpeg	mtrt
s1	17.4	33.0	6.7	4.3	30.2	58.2
s10	7.7	49.8	12.6	17.3	25.4	64.8
s100	12.3	48.6	12.6	18.0	30.2	65.0

6.1　逐次実行を省略する能力

われわれのデータ再利用機構は，バイトコードメソッド処理を，JVM命令の通常の逐次実行で処理する他に，RBヒットに基づいて，ヒットエントリからの出力データのメモリ書き出しで簡便に処理することもできる．ネイティブメソッド処理およびネイティブメソッド処理を内部に含む処理は，RBに登録されない．RBに登録される処理は，バイトコードメソッド処理であり，バイトコードメソッド処理全体は，RBが本来的に扱い得る処理全体を示す．バイトコードメソッド処理は，場合によっては，親子関係にある複数のメソッド処理をまとめた形と，一つのメソッド処理とで，重複してRBに登録される．RBに登録されたバイトコード処理は，後に，同じバイトコード処理を開始する場合にRBヒットされる．このときには，このバイトコード処理は，ヒットエントリからの出力データのメモリ書き出しで簡便に処理される．

本節では，データ再利用機構を搭載したわれわれの高速JVMが，通常のメソッド逐次処理を省略し，必要なメソッド処理を，RBヒットに基づいたヒットエントリからの出力データのメモリ書き出しで代替する能力を調べた．SPEC

JVM98 の実行オプション s1，s10，s100 で測定した結果が，それぞれ表5〜表7である．大きな値のデータは紙幅の都合上下3桁を「k」で省略している．これらの表の読み方について，データ再利用の仕組みをふまえて以下にまとめる．ただし以下のデータは，全て，ネイティブメソッドコールを除外したデータである．

第一列

- ネイティブメソッドコールを除いた，バイトコードメソッドコールの総数．ネイティブメソッドの処理は再利用しないので，RB が本来的に扱うべき処理全体と言える．

第二列

- RB 機構により省略に成功したメソッドコール数（上段）．即ち，RB ヒット数にヒット当たりの平均省略メソッド数を乗じた値である．
- バイトコードメソッドコールの総数を全体とした，省略に成功したメソッドコール数の割合（下段）．

第三列

表5: RB の性能評価 (s1)

s1	メソッドコール総数	省略メソッド数 省略率	実行命令数	省略命令数 省略率	ヒット回数	平均省略メソッド数 平均省略命令数
comp	17358k	3936k	955125k	35672k	3935k	1.000
		22.68%		3.73%		9.064
jess	595k	196k	9295k	1113k	190k	1.028
		32.98%		11.98%		5.831
db	105k	10k	2033k	101k	8k	1.137
		9.58%		5.00%		11.398
javac	635k	31k	10104k	165k	30k	1.055
		5.02%		1.63%		5.466
mpeg	1138k	420k	116449k	2991k	418k	1.005
		36.94%		2.57%		7.155
mtrt	6083k	3641k	68895k	12014k	3630k	1.003
		59.85%		17.44%		3.309

表6: RB の性能評価 (s10)

s10	メソッドコール総数	省略メソッド数 省略率	実行命令数	省略命令数 省略率	ヒット回数	平均省略メソッド数 平均省略命令数
comp	18198k	3239k	1138455k	30274k	3238k	1.000
		17.80%		2.66%		9.349
jess	6077k	3535k	123097k	28994k	3528k	1.002
		58.17%		23.55%		8.218
db	2257k	596k	77328k	5206k	575k	1.037
		26.43%		6.73%		9.051
javac	4657k	839k	80617k	3188k	813k	1.031
		18.02%		3.95%		3.917
mpeg	4689k	1538k	611162k	11043k	1531k	1.005
		32.81%		1.81%		7.211
mtrt	19872k	14941k	177118k	49603k	14844k	1.007
		75.19%		28.01%		3.342

表7: RB の性能評価 (s100)

s100	メソッドコール総数	省略メソッド数 省略率	実行命令数	省略命令数 省略率	ヒット回数	平均省略メソッド数 平均省略命令数
comp	45246k	10987k	2495922k	100037k	10986k	1.000
		24.28%		4.01%		9.105
jess	124134k	58879k	1960888k	380616k	58846k	1.001
		47.43%		19.41%		6.468
db	139170k	32915k	3971094k	245276k	32417k	1.015
		23.65%		6.18%		7.566
javac	158734k	30270k	2606976k	114779k	29498k	1.026
		19.07%		4.40%		3.891
mpeg	108475k	40503k	11490256k	283798k	40434k	1.002
		37.34%		2.47%		7.019
mtrt	282961k	214050k	2137179k	692380k	212602k	1.007
		75.65%		32.40%		3.257

- データ再利用を用いなかった時に，バイトコードメソッドコールで実行する総JVM命令数.

第四列

- RB機構により省略に成功したJVM命令数（上段）．ただし，それぞれのJVM命令は1〜5バイトまでの可変長の大きさを持つ，
- 総JVM命令数を全体とした，省略に成功したJVM命令数の割合（下段）.

第五列

- RBヒットの回数．即ち通常の逐次処理の省略に成功した回数.

第六列

- RBヒット毎の，省略メソッドコール数の平均値（上段）．すわなち，第二列上段に示した省略メソッド数を，第五列に示したヒット回数で除した値である.
- RBヒット毎の，省略に成功したJVM命令数の平均値（下段）．すわなち，第四列上段に示した省略JVM命令数を，第五列に示したヒット回数で除した値である.

表5〜表7から，データ再利用機構付き高速JVMは，次のような特長を持つことがわかる.

i. 各ベンチマーク実行で省略メソッド数の割合と省略JVMコード長の割合では，後者の方が小さな値となる場合が多い．これから，RBを用いた再利用が成功するメソッド処理単位（再利用単位）は，全体の平均と比べると小さいコード長のメソッドから構成されていることがわかる.
ii. 省略メソッド数の割合については殆どのベンチマーク実行において1割以上の値をとり，場合によっては30%〜70%という高い値を示す．省略JVM命令の割合においても殆どのベンチマーク実行において5%以上の値をとる．場合によっては20%から70%近い高い値を示している.
iii. 1度の再利用処理で省略されるメソッド数は1.000〜1.137個であり，殆どの場合は1.01個以下である．これはネストしたメソッド実行の再利用が成功する場合は，かなり限定的なアプリケーションに対してのみであることを示している.
iv. データ再利用は，ベンチマークによる効果の差が大きく，その点が大きな問題点である．ベンチマークによる性能の不安定性の他に，実行オプションによる性能のばらつきも大きく，安定的な性能発揮という点で問題が多い.

以上より，われわれの確立した，RB付きJVMにおける，メソッドを単位としたデータ再利用方式は，高速なJVMの実現に大きな威力を発揮することが明らかである．しかし，また，ネストしたメソッドの再利用は殆ど成功しておらず，どのようなプログラムの実行に対しても常に有効であるという性能の安定的にも課題が存在することがわかる.

ここで，各メソッドが含むJVM命令系列のコード長（バイト数）に着目して，実行オプションs100における実行について行った詳細な分析結果を図12〜図13に示す．横軸にはメソッドのコード長を測った値をとり，縦軸には処理回数をとっている．ただし縦軸・横軸は共に対数表示を行っている．各ベンチマークの3つの図で，上段はRBを備えない通常のJVMでメソッド処理全体についての統計を計測したものである（図9と同じデータを，比較のために再掲した）．中段の図については6.3節で説明する．下段の図はRB機構を搭載したJVMで計測した値であり，RBヒットに基づいて，ヒットエントリの出力データからのメモリ書き出しを行い，逐次実行の省略に成功したメソッド処理の統計である．これらの図より，次のような考察を行うことができる.

i. コードサイズの大きなメソッドの処理が，RBヒットに基づく逐次実行省略の対象となることは少ない.
ii. とりわけ100バイト程度を超えるコード長のメソッドの処理は，殆ど省略の対象となっていない.
iii. これは，表7において，実行命令数での省略率が，メソッド処理数での省略率よりも低かった事実と符合する.
iv. 少数のメソッドに関わるメソッド処理の省略が，省略全体の大部分を占める傾向が強い．これらのメソッドに特化して対処を図ることが枢要な課題であることが分かる.

6.2 再利用を行う処理単位

われわれは，先に，データ再利用を行う処理単位の構成について子細な検討を行い，メソッドの実行を単位として再利用を行う方式を採用する理由を述べた（第3章節，第4章節）．ここでは，われわれの採用した，メソッドを単位としたプログラム単位の持つ性質を，含有するメソッドの個数と，単位処理への入出力の観点について調べた結果をまとめた．以下では，RB登録時点とRB

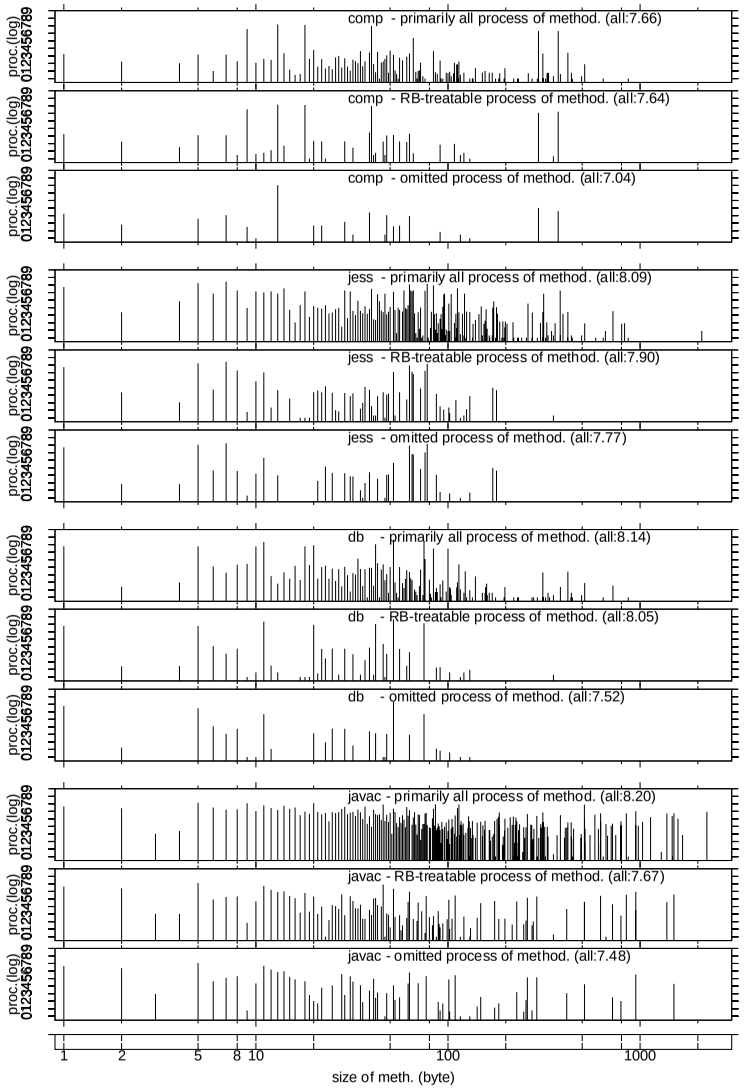


図12: コードサイズの観点から見たメソッド処理の分析 (comp, jess, db, javac)

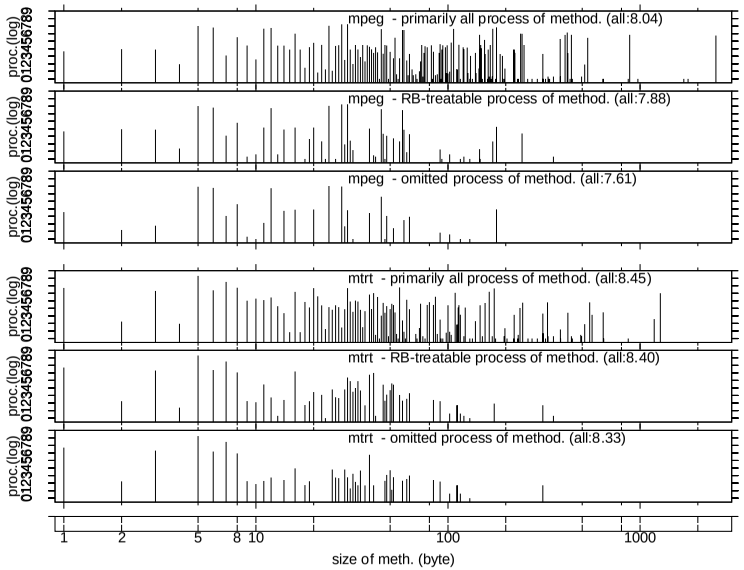


図13: コードサイズの観点から見たメソッド処理の分析 (mpeg, mtrt)

ヒット時点で統計値をとっている.

6.2.1 含有するメソッドの個数，最大ネスト数

われわれは先に，RB による各再利用処理の実行が含有するメソッド数が平均で 1.000〜1.137 と小さいことを見た．殆どのベンチマーク実行の結果では，1.01 以下であった（表5〜表7）．ネストして呼び出される複数のメソッドの処理を一括して再利用することは，殆どできていない．これは実行オプション s1 で調べたものであるが，実行オプション s10 の場合の結果は，おおむね，各ベンチマークで，s1 の場合と s100 の場合の中間的な結果であった．表 9 の各ベンチマーク欄で，上段は RB 登録されたエントリのデータについて，下段は実際のヒットエントリのデータについて，それぞれ，本来的に含有していたメソッドの個数を調べている．ただし，空欄の場合は，そこに属する RB エントリの登録またはヒットが，全く存しなかったことを示している．これらの表より次のような考察が可能である.

i. 登録時点の平均値は 1.000〜1.873，ヒット時点での平均値は 1.000〜1.137

表8: 処理単位が含有するメソッドの数

[上段：RB登録された処理について
下段：再利用に成功した処理について]

s1	メソッド数						
	平均	1	2	3	4	5	6
comp	1.000	99.997%	0.001%	0.001%	0.001%	0.000%	
	1.000	99.971%	0.029%				
jess	1.293	81.600%	10.023%	6.037%	2.186%	0.151%	0.003%
	1.028	97.227%	2.707%	0.064%	0.003%		
db	1.873	56.148%	15.632%	13.763%	13.651%	0.806%	
	1.137	86.321%	13.679%				
javac	1.022	98.362%	1.309%	0.138%	0.111%	0.080%	
	1.055	94.471%	5.529%				
mpeg	1.183	81.916%	17.910%	0.089%	0.081%	0.004%	
	1.005	99.495%	0.481%	0.004%	0.020%		
mtrt	1.056	95.611%	2.861%	0.037%	1.038%	0.001%	
	1.003	99.709%	0.291%				

である．殆どのベンチマーク実効では，登録時点での平均値の方が大きい．

ii. compress，javac および mtrt の各ベンチマークでは，登録ならびにヒットしたRBエントリの中で，おおむね95%以上のエントリデータが，単一のメソッドの実行結果であった．

iii. jess，db および mpeg の各ベンチマークについては，複数のメソッドの実行結果となるエントリデータの取り扱いの割合も，登録時の統計では8〜44%に達する．

iv. しかし，これらのベンチマーク実行の場合でも，含有するメソッド数が3個以上である場合は少ない．また，ヒット時点での統計では，単一のメソッドの実行結果であるエントリのヒットは，db ベンチマークの s1 オプションの場合以外では，95%以上となってしまう．

v. したがって，jess，db および mpeg の実行で一定数がRB登録された，複数のメソッドの実行結果となるエントリデータは，少数のヒットしか受けな

表9: 処理単位が含有するメソッドの数

[上段：RB登録された処理について
下段：再利用に成功した処理について]

s100	メソッド数						
	平均	1	2	3	4	5	6
comp	1.000	99.998%	0.001%	0.001%	0.000%	0.000%	
	1.000	99.989%	0.011%			0.000%	
jess	1.080	92.424%	7.193%	0.368%	0.015%	0.001%	0.000%
	1.001	99.947%	0.050%	0.003%	0.000%	0.000%	
db	1.162	83.862%	16.125%	0.007%	0.007%	0.000%	
	1.015	98.462%	1.538%		0.000%		
javac	1.025	97.872%	1.829%	0.259%	0.033%	0.007%	0.001%
	1.026	97.451%	2.483%	0.063%	0.002%	0.000%	0.000%
mpeg	1.181	81.963%	17.989%	0.028%	0.019%	0.001%	
	1.002	99.870%	0.109%	0.002%	0.019%		
mtrt	1.079	96.149%	0.521%	0.005%	2.662%	0.000%	0.000%
	1.007	99.321%	0.679%	0.000%			0.000%

かったり，または，全く無意味に上書き・廃棄されてしまったことが推定される．

再利用を行う JVM 命令系列を決めるにあたって，メソッドコールがネストした処理単位の再利用を行わないことが考えられるその場合のRB機構の簡素化がもたらす，RB機構の効率化は，同時に複数のRBエントリへの登録を行う処理が省略できることである．ソフトウェア上のインプリメントでは，このメリットはそれ程には大きくないが，ハードウェアでのインプリメントにおいては，ハードウェア量のかなりの低減がもたらされると考えられる．

一方，ネストしたメソッドに跨る命令系列の再利用処理が行えていない理由は何処にあるか，現在のところはよく分かっていない．ネストしたメソッドの有効な再利用処理を行うためには，JVM バイトコード実行におけるメソッドコールのネストの性質の，さらに詳しい検討が必要である．

6.2.2 入出力の大きさ

RB エントリの入力データは現在のメモリ状態と比較され，一致した場合には同エントリの出力データがメモリに書き出される．このとき，われわれの確立した方式では，フレーム上のメソッド引数，およびヒープ上のスタティック・ローカル変数，ならびに配列の比較を行う．これらの個数の平均値の統計が表 10 の第3〜第5列である．各 RB エントリとの比較の手間は第3列の値と正の相関関係を持つ．各ベンチマークで上段は登録エントリについて，下段はヒットエントリについて，それぞれ統計をとった．入力が一致する RB エントリが存在する時には，そのエントリに登録された出力情報について，返り値を呼出し元フレームへ，スタティック・ローカル変数ならびに配列の出力を，ヒープ上の所定のデータ域に，それぞれ書き出す．これら，出力の個数の平均値に関する統計が，表 10 の第6〜第8列である．

表 10 より次のような考察を行うことができる．

表 10: 再利用を行う処理単位への入出力の大きさ

[上段：RB 登録された処理について
下段：再利用に供された処理について]

s100		入力			出力		
	合計	小計	フレーム	ヒープ	小計	フレーム	ヒープ
comp	7.219	5.067	1.779	3.288	2.152	0.430	1.722
	5.012	4.010	1.999	2.012	1.002	0.999	0.002
jess	4.362	3.371	1.761	1.610	0.991	0.893	0.098
	3.907	2.995	1.645	1.350	0.912	0.912	0.000
db	7.367	6.172	1.611	4.561	1.195	0.999	0.196
	4.890	4.074	1.724	2.350	0.816	0.816	0.000
javac	5.124	4.158	1.853	2.305	0.966	0.351	0.615
	2.761	1.920	1.311	0.608	0.841	0.835	0.006
mpeg	5.211	3.618	1.537	2.081	1.593	0.973	0.620
	3.639	2.403	1.564	0.840	1.236	0.574	0.662
mtrt	3.639	2.476	1.277	1.199	1.163	0.928	0.234
	3.224	2.248	1.151	1.097	0.976	0.969	0.007

i. 再利用単位系列への入出力の個数は，入力の方が多い．とりわけベンチマーク db と javac の場合の出力個数は 1 以下の値であり，出力が 0 個の RB エントリが，相当数扱われていることがわかる．

ii. 入力の個数の多さは一般的に，単位系列への入力値の局所性を減少させると推定される．表 10 の第三列に示した入力データ個数の合計は各ベンチマークで全て，下段のヒット時における値は上段の登録時の値よりも小さい．上の推定は正しいことがわかる．しかし，フレームとヒープでの値を個別に見ると，必ずしも広くあてはまる傾向ではない．

iii. 各ベンチマークのフレームとヒープの入出力欄，計 4 個のデータについて，上段の登録時の値と下段のヒット時の値との間に倍以上の開きが存する場合がある．そのような性質が発生する例は比較的ヒープへの出力の個数の平均値に多いようである．

iv. ヒープへの出力の個数は，殆どの場合 0 である．とりわけ，ヒット時の統計にその傾向が大きい．

v. フレームへの出力（すなわち返り値）の平均値は，ヒット時点では，殆どの場合に 0.9 以上の値をしめしている．返り値の書き込みのない処理と比べて，返り値の書き込みの存在する処理の再利用に，多く成功していることが分かる．

入力の平均個数は 2〜4 個と大きくない値である．とりわけヒット時の統計の場合に小さな値をとる傾向があるために，値の局所性（類似性）に基づいて入力データのデータ投機を行い，RB のエントリを投機的に用意する機構 [31] が有効に働く必要条件を満たしていると考えられる．

6.3 RB が扱い得る処理

われわれは先に，再利用困難な JVM 命令について検討を加え，それらの命令を含むメソッドの再利用を行わないことを決めた．ネイティブメソッドを対象とした `invoke` 命令により，ネイティブコードの処理が行われる場合も同様である．また，RB は，表 3 で示した各 RB エントリの大きさを溢れる入出力の個数が発生すると，その時点での登録経過を無効化する[1)]．これらの問題を包まない，RB が扱い得るメソッド処理の割合が十分に高いことは，RB が大規模な

[1)] ただし，表 3 に示した RB エントリの大きさは，表 5〜表 7 などに示した性能に大きな影響を及ぼさない限度で，最も小さな RB となるように設定したものであった．

逐次処理省略を達成するための必要条件である．RBが扱い得る処理の一部は，RBに登録された過去の実行結果のデータに基づき，バイトコードの逐次実行に代えて，ヒットエントリの出力データのメモリ書出しで済まされる．

RBが扱い得るメソッド処理全体の量を測定するために，逐次処理の省略を一切行わないが，処理データの登録は行うベンチマーク実行を試みた．このような状態で登録の対象となったメソッド処理は，RBが扱い得るメソッド処理全体に他ならない．そのような処理の幾らかの部分が，RBヒットに基づいた逐次処理の省略を行う実行の場合に，省略の対象となる．RBが扱い得るメソッド処理は，全てのメソッド処理の中で，有効なRB登録を行い得た処理の割合と，RBヒットに基づく再利用実行により省略されたメソッド処理の割合との和に他ならない．

表11は，RBが扱い得るメソッド処理の割合（上段），および，その中で，逐次処理の省略に成功したメソッド処理の割合[1)]（下段）をまとめたものである．さらに，実行オプションs100の処理に対して，メソッドのコード長に注目して，RBが扱い得るメソッド処理の詳細な分析を行った．この結果が，先に6.1節で示した図12〜図13における各ベンチマーク中段の図である（39〜13ページ）．これらの図表より，次のような考察が可能である．

i. 再利用可能な処理の割合は，16.7%〜88.93%とばらつきが広い．compress ベンチマーク以外では，実行オプションによるばらつきも大きい．

ii. われわれは，先の5.3節でネイティブメソッドコールの量に関する分析を行い，jessおよびdbベンチマークのオプションs1実行，ならびにjavacベンチマークの実行ではネイティブメソッドコールの量が相対的に多いことを知っている（表2の浮き出し欄）．表11においてこれらのベンチマーク実行では明らかにRBが扱い得るメソッド処理の割合が小さく（表11の浮き出し欄），省略されたメソッド処理の割合も相対的に低い．ネイティブメソッドの増加が甚大な影響を及ぼしていることがわかる．

iii. RBが扱い得る処理の割合が高いほどに，RBヒットに基づいて逐次実行を省略されたメソッド処理の割合も高くなるという傾向がある．しかし，compressベンチマークでは，例外的に，RBが扱い得る処理の割合が80%に達しているにもかかわらず，RBヒットに基づいて逐次実行の省略に成功したメソッド処理の割合は，17.96%〜24.58%と低い．これは，再利用可能な処理の中の冗長性が低いか，または，冗長性の抽出に失敗しているためであると考えられる．

iv. 6.1節において，われわれは，コード長の大きいメソッドの処理は，逐次実行の省略を行うことが困難であることを知った．図12〜図13における各メソッド中段の図によれば，RBへの登録さえ行えていなかったことが分かる．

v. 図12〜図13によれば，およそ，RBヒットに基づいて逐次実行を省略できるメソッド処理は，RBが扱い得る処理の範囲に大きく依存していることがわかる．

表11: メソッド処理とRB

[上段：再利用可能な処理の割合
下段：RBヒットにより省略された処理の割合]

	comp	jess	db	javac	mpeg	mtrt
s1	95.3	48.5	27.3	18.8	67.6	70.5
	22.7	33.0	9.6	5.0	36.9	59.9
s10	95.1	68.2	74.3	28.6	70.0	84.7
	17.8	58.2	26.4	18.0	32.8	75.2
s100	95.4	64.0	80.1	29.3	69.5	89.0
	24.3	47.4	23.7	19.1	37.3	75.6

[1)] 表5〜表7の第二列下段からの再掲

6.4 作成される個々のRBエントリ

6.1節で明らかとなったように，RBは，バイトコードメソッド処理の相当部分をRBエントリからの出力データのメモリ書き出しで処理する．また，6.3節で示したように，残されたバイトコード逐次実行処理の相当部分は，RBが扱い得る処理であり，その処理結果はRBエントリに有効に登録される．

表12の上段のデータは，RB上に作成された個々のエントリデータが，他のデータからの上書きまたはプログラム終了によって廃棄されるまでの間に，何

度のRBヒットを受けるかについて，その平均値[1]を調べたものである．ここで0.2〜7.9という値は，価値の少ない処理データのRB登録が多数行われていることを示すと考えられる．そこで，作成される個々のRBエントリに対するヒット回数の詳細を実行オプションs100の場合について調べると，図14のようになった．この図によれば，1000回以上ヒットするエントリが存在する一方で，大部分のRBエントリは一度もヒットされない．一度もRBヒットされないエントリの登録は全くのオーバヘッドに他ならないが，一度もRBヒットされないエントリは全体の大部分を占めることが分かる．実際にも，作成される全てのRBエントリの中で一度もRBヒットされなかったものの割合を調べると表13のようになった．殆どのベンチマーク実行において60%を超えるRB登録が結果的に無駄な登録となっている．

ここで6.1節での考察によると，RBヒットの対象となるメソッドは限定的であり，そのようなメソッドを静的・動的にある程度予測することはそれほど難しくないと予想される．実際にも，例えば，記憶領域（メソッドエリア）に展開される，全ての使用バイトコードメソッドを全体として，RBが扱い得る処理を構成するメソッドの割合と，その中で，データ再利用によって実際に逐次実行を省略した処理を構成するメソッドの割合を調べて，それぞれ表14の上段・下段の値を得ている．RBが扱い得る処理の中で，RB登録を行っても一度もヒットの対象とならないメソッドとそうでないメソッドが，ある程度明確に分かれているものと考えられる．なお，表12で各ベンチマークの下段の値は，一度もヒットの対象とならなかったエントリを除いた場合の平均ヒット回数であり，飛躍的な上昇がもたらされる．

以上の考察より，予測・投機アルゴリズムまたは事前の実行解析などによってヒットの可能性が低いエントリの登録を排除する機構が，極めて有効であると考えられる．そのような機構により，メソッドの省略率などの性能を低下させることなく，RB登録および検索の手間を低減させることができる．また，小さなRBでも有効な動作が可能なデータ再利用が実現できるようになり，RBのハードウェア量を削減するためにも重要なことである．

[1] 即ち，総ヒット数 / 総登録数．

表12: 作成したRBエントリの平均ヒット回数
［下段は全くヒット対象とならなかったエントリを除いた場合］

	comp	jess	db	javac	mpeg	mtrt
s1	0.31	2.07	0.49	0.35	1.20	5.63
	6.16	9.45	9.50	7.52	11.70	16.70
s10	0.23	5.79	0.80	1.66	0.88	7.88
	6.75	17.15	2.82	13.16	10.47	18.56
s100	0.34	2.85	0.57	1.81	1.16	5.63
	6.56	8.53	3.10	16.36	11.50	12.83

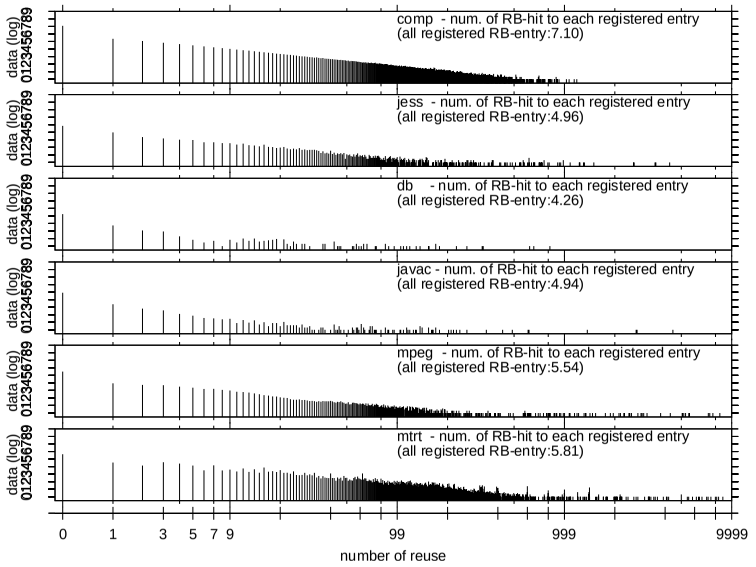


図14: 作成したRBエントリのヒット回数

表13: 作成するRBエントリの中で，全くヒット対象とならないものの割合(%)

	comp	jess	db	javac	mpeg	mtrt
s1	94.9	78.1	94.9	95.4	89.7	66.3
s10	96.6	66.2	71.7	87.4	91.6	57.5
s100	94.8	66.6	81.6	88.9	89.9	56.2

表 14: メソッドと RB

上段：RB が扱い得る処理に関わるメソッド数 (%)
下段：RB に省略された処理に関わるメソッド数 (%)

	comp	jess	db	javac	mpeg	mtrt
s1	25.2	27.5	24.8	27.0	24.2	32.6
	6.2	9.1	7.3	10.6	9.7	14.4
s10	25.2	27.3	25.2	30.0	24.8	33.0
	5.9	8.6	7.4	14.3	10.0	15.4
s100	25.6	27.7	25.0	33.4	25.2	33.3
	6.7	9.2	7.5	17.0	9.9	15.6

第 7 章　おわりに

本稿では，メソッドを単位とした JVM におけるデータ再利用機構を，SPEC JVM98 ベンチマークを題材として詳細な検討を行った．JVM にわれわれのデータ再利用機構を適用することにより，メソッド処理数で 19〜76%，実行命令数で 2〜30% もの処理が省略されるなど大きな高速化をもたらすことが明らかとなった．しかし，入れ子構造のメソッドを全体的に再利用することは困難であり，ネイティブメソッドも場合によっては相当程度の障害となり得ることが分かった．一方で入力データの平均個数は高々2〜4 個に過ぎないことが判明したために，入力データの予測により投機的に RB エントリを作成することも有効であると考えられる．

現在のわれわれの機構では大量の RB 登録を行ってエントリを作成するが，その中の 60〜90% のエントリは一度もヒットの対象となることがない．しかし，省略される処理に関わるメソッドの個数が，RB 登録に関わるメソッドの半数以下に過ぎないなど，RB 登録を行っても一度もヒットの対象とならない処理とそうでない処理がある程度明確に分離できる見込みがある．さらに，省略される逐次処理に関わるメソッドの中で，特定の少数のメソッドに関わる処理の省略が全体の省略の大部分を占める傾向も明らかとなっている．本稿に示した結果などを基として，大部分の RB ヒットに関わるメソッドを，静的・動的に，ある程度予測することはそれほど難しくないと考えられる．

今後，まず，集中的に RB ヒットを受けるメソッド処理の，メソッド・入力データに関するより具体的な規則性を発見するべくさらに精緻な分析を行い，将来の RB ヒットの可能性に基づく，投機的な機構の研究を行うことが，今後の課題となっている．選択的な RB 登録，ならびに事前の RB データ作成などの機構を構築し，格段に有為な高速化手法へとわれわれの RB 方式を発展させることができると考えている．

謝辞

本研究の機会を与えてくださった，富田 眞治教授に深く感謝の意を表します.

日頃から御討論下さるコンピュータ工学講座計算機アーキテクチャ分野の諸氏に感謝致します.

参考文献

[1] Tim Lindholm and Frank Yellin: *The Java^{TM} Virtual Machine Specification*, Addison-wesley Publishers (1997).

[2] A. Goldberg and D. Robinson: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley (1983).

[3] B. Stroustrup: *The C++ Programming Language, 1st ed*, Addison-Wesley (1986).

[4] 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).

[5] P. J. Koopman: *Stack Computers:the new wave*, Ellis Horwood Ltd. (1989).

[6] R. Greenblatt et al.: *The LISP Machine in Interactive Programming Environments (ed. D.R.Barstow)*, McGraw-Hill (1986).

[7] I. Takeuchi et al.: A concurrent multiple-paradigm list processor TAO/ELIS., *Proceedings 1987 Fall Joint Computer Conference – Exploring Technology: Today and Tomorrow*, pp. 167–74 (1987).

[8] E. Goto et al.: Design of a Lisp chip into a system for military AI, *Electronics*, pp. 95–96 (1987).

[9] 金田悠紀夫: Prolog マシン, 森北出版 (1992).

[10] 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).

[11] 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. 113, No. 3, pp. 17–24 (1995).

[12] L. P. Deutsch and A. M. Schiffman: Efficient Implementation of the Smalltalk-80 System, *Proceedings, 11th Annual ACM Symposium on the Principles of Programming Languages*, pp. 297–302 (1984).

[13] Gaudin, S.: Sun hits Java punch list, but users crave speed, *Computerworld*, Vol. 31, No. 29, pp. 10–10 (1997).

[14] B. Alpern, C. R. Attanasio, J. J. Barton et al.: The Jalapeño virtual machine, *IBM Systems Journal*, Vol. 39, No. 1, pp. 211–238 (2000).

[15] M.Lipasti and J. Shen: Exceeding the Dataflow Limit via Value Prediction,

Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO-29), pp. 226–237 (1996).

[16] G. Tyson and T. Austin: Improving the Accurancy and Performance of Memory Communication through Renaming, In MICRO-30 [32], pp. 218–227.

[17] A. Sodani and G. S. Sohi: Understanding the Differences Between Value Prediction and Instruction Reuse, *31st Annual ACM/IEEE International Symposium on Microarchitecture* (1998).

[18] K. Wang and M. Franklin: Highly Accurate Data Value Prediction using Hybrid Predictors, In MICRO-30 [32], pp. 281–290.

[19] Y. Sazeides and J. Smith: The Predictability of Data Values, In MICRO-30 [32], pp. 248–258.

[20] A. Sodani and G. S. Sohi: Dynamic Instruction Reuse, *24th Annual International Symposium on Computer Architecture*, pp. 194–205 (1997).

[21] A. González, J. Tubella and C. Molina: Trace-Level Reuse, *1999 International Conference on Parallel Processing (CPP'99* (1999).

[22] Daniel A. Connors and Wen-mei W. Hwu: Compiler-Directed Dynamic Computation Reuse; Rationale and Initial Results, *32nd Annual International Symposium on Microarchitecture (MICRO'99)* (1999).

[23] J. Huang and D. J. Lilja: Extending Value Reuse to Basic Blocks with Compiler Support, *IEEE Transactions on Computers*, Vol. 49, No. 4, pp. 331–347 (2000).

[24] L. R. Ton et al.: Instruction Folding in Java Processor, *1997 International Conference on Parallel and Distributed Systems* (1997).

[25] J. Michael O'Connor and Marc Tremblay: picoJava-I: the Java Virtual Machine in hardware, *IEEE Micro*, Vol. 17, No. 2, pp. 45–53 (1997).

[26] 重田大助, 小川洋平, 山田克樹, 中島康彦, 富田眞治: 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, 情報処理学会論文誌:ハイパフォーマンスコンピューティングシステム HPS1, Vol. 41, No. SIG5 HPS1, pp. 13–18 (2000).

[27] James Gosling, Bill Joy and Guy Steele: *The Java^{TM} Language Specification*, Addison-wesley Publishers (1997).

[28] The Unicode Consortium: *The Unicode Standard, Version 2.0*, Addison-Wesley Pub Co. (1996).

[29] Evaluation Corporation: SPEC JVM98 VERSION 1.03, the Standard Performance (1998). http://www.spec.org/osg/jvm98/.

[30] 重田大介, 小川洋平, 山田克樹, 中島康彦, 富田眞治: 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, 情報処理学会計算機アーキテクチャ研究会報告, Vol. 80, pp. 13–18 (2000).

[31] 山田克樹, 中島康彦, 富田眞治: 投機的手法を用いたデータ再利用による Java 仮想マシンの高速化, 情報処理学会研究報告 ARC, Vol. 139, No. 29, pp. 169–174 (2000).

[32] IEEE Computer Society TC-MICRO and ACM SIGMICRO: *Proceedings of the 30th Ann. International Symposium Microarchitecture (MICRO-30)* (1997).