実行結果再利用による
Javaバイトコード実行の高速化手法

重田 大助

平成12年2月9日提出

Abstract

Javaプログラミング言語はプログラミングが容易であり，実行環境がプラットフォームに依存せず，しかも高い安全性を持つという特徴により，今後ますます使用される範囲は広がっていくものと思われる．実行基盤であるJavaバーチャルマシンは，型付けが強く，また実行時に実行されるメソッドが決定されるという特徴を持っているため，Javaコンパイラはコンパイル時の最適化を十分に行うことができない．また，Javaバーチャルマシンはスタックマシンであるため，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．

本稿ではJavaバイトコード実行時の高速化手法としてデータ再利用を提案した．本手法は実行結果を保存しておき，再度同じデータを参照して実行する場合に実行結果を再利用するものである．データ再利用を行う単位として，区切り命令間の命令群およびメソッド実行時の命令群という2つの方法を提案した．各手法を，インタプリタ方式によるJavaバーチャルマシンに実装した結果，データ再利用の効果と比較してそのオーバヘッドが大きいため，実際に高速化することはできなかったものの，Javaバイトコードの性質がデータ再利用に適していることが判明した．そこで，ハードウェアにおいてデータ再利用を実行することにより，そのオーバヘッドをバイトコードの実行時間に隠蔽した．

メソッドを単位としたデータ再利用の手法をハードウェアへ実装することにより，メソッドの呼び出しおよびリターンという処理に伴うパイプラインハザードを減少させることが可能となることを示した．削減されたサイクル数は0.1%から47.1%とばらつきが存在する．使用される浮動小数点数の種類は整数と比較して膨大となりやすい．よって，整数を扱うプログラムは浮動小数点数を扱うプログラムよりも，データ再利用の効果が高くなる．また，一般的に，オブジェクト指向を強く意識して書かれたプログラムは，実行するメソッド数が増加するため，そうでないものと比較して多くの実行時間を必要とする．しかし，メソッドの実行中にヒープ領域を参照する回数は少なくなり，データ再利用の効果が高くなる．データ再利用を用いることにより，オブジェクト指向を強く意識したプログラムが高速に実行できることを示した．

The Java programming language is simple, robust and hardware independent. So, this language will come into wide use. Java Virtual Machine is the basis of this language. The followings are the features of Java Virtual Machine: typing are tight; which method is executed is determined dynamically. So, Java compiler cannot generate optimized program code. And, Java Virtual Machine is a stack machine. So, if it executes bytecode as they are described, it must move data in vain.

We propose Data Value Reuse executing Java bytecode at high speed. Data Value Reuse is the technique to reuse the results of instructions when input data are the same. There are two ways of reusing traces: reusing the instruction traces which are divided by the special instructions, reusing those are same as method calls. As each way, we implemented Data Value Reuse on Java bytecode interpreter. Consequently, the execution time to apply Data Value Reuse is longer than that is reduced by applying this technique. But, the nature of Java bytecode is suited to Data Value Reuse. So, we implemented Data Value Reuse on a hardware implemented Java Virtual Machine to conceal the overhead to apply this technique in the bytcode execution time.

Hardware implemented Data Value Reuse in reusing method executions can reduce pipeline hazards resulting from method call and return. Reduced cycles vary in the rate from 0.1 to 47.1. The number of values of floating point number is more than that of integer number. In Data Value Reuse programs to deal integer values is more effective than those to deal floating point values. And, the number of method calls in programs described in consideration of object oriented programming model is larger than that in programs which is not so. But, the number of heap accesses in the former programs is smaller than that in the latter programs. As a result, we show that Data Value Reuse in the former programs is more effective than that in the latter programs.

目次

1 はじめに

Javaプログラミング言語はプログラミングが容易であり，実行環境がプラットフォームに依存せず，しかも高い安全性を持つという特徴により，今日ではJavaプログラミング言語はインターネット上で広く使用されている．今後は，ますますその使用される範囲は広がっていくものと思われる．Javaプログラミング言語の実行の基盤となるJavaバーチャルマシンは，型付けが強く，また実行時に実行されるメソッドが決定されるといった特徴を持っている．このため，Javaプログラミング言語からバイトコードを生成するJavaコンパイラ

は，コンパイル時の最適化を十分に行うことができない．Java バーチャルマシンはスタックマシンであるため，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．

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，Java バーチャルマシンが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．この手法は，一般の RISC プロセッサよりも，前後の命令間でのデータ依存関係の強いスタックマシンである Java バーチャルマシンに適用するのに適している．まず，Java バイトコードにより記述されたベンチマークプログラムを対象として，データ再利用を適用するために必要な処理および適用可能な範囲に関する調査を行った．そして，この結果に基づいて，Java バイトコードの特性を最大限に生かすデータ再利用のためのハードウェア機構を仮定し，性能予測を行なった．

以下，2においては，背景として，Java プログラミング言語を実行する基盤である Java バーチャルマシンの特徴からその問題点と解決法について述べる．3においては，Java バーチャルマシンにおけるデータ再利用の実装方法について述べ，その時の問題点と解決法を示す．4においては，Java バーチャルマシンをハードウェア化した場合のデータ再利用の適用方法について述べる．5においては，データ再利用の効果を示し，考察を行う．

2　研究の背景および課題

2.1　Java プログラミング言語

Java プログラミング言語は，並列駆動が可能なクラスを基本としたオブジェクト指向型プログラミング言語である．Java プログラミング言語は，データ型を標準化するなど，可能な限り実装に依存する部分がないように設計されている．作成したプログラムコードはアーキテクチャに依存しないため，アプリケーション開発者がプログラムをいったん Java プログラミング言語で書けば，それはどのような環境においても実行可能なものとなる．また，Java プログラミング言語で作成されたコードはプログラムサイズが非常に小さいという特徴を持っている．

Java プログラミング言語はポインタ機能を排除している．明示的な領域解放にまつわる安全性の問題には，ガーベッジ・コレクタの採用による自動記憶管理を行っている．さらに，コンパイルおよび実行時に型のチェックなど厳しいエラーチェックを行い，メモリの不正使用などのバグの原因を早期に検出することが可能となっている．これらのことにより，Java プログラミング言語は，高い安全性を実現している．

Java プログラミング言語では，オブジェクトを単位として効率良く開発することが可能である．また，言語レベルでマルチスレッド機能をサポートしているため，スレッドを用いたプログラムの記述が容易である．Java プログラミング言語はネットワーク経由でプログラムを実行することが可能である．TCP/IP 用のライブラリも用意されているため，ネットワークで結合された現在の計算機環境に Java プログラミング言語は非常に適している．

このように，プログラミングが容易であり，プラットフォームに依存せずに実行でき，しかも安全性が高いという特徴により，今日では Java プログラミング言語はインターネット上で広く使用されている．今後は，ますます利用範囲が広がっていくものと思われる．

2.2　Java バーチャルマシン

Java バーチャルマシンは Java プログラミング言語の土台となるものであり，マシンに依存しないコードにコンパイルされたプログラムを実行するためのものである．

Java バーチャルマシンは，特定のファイル・フォーマット，すなわちクラスファイル・フォーマットの情報のみに基づいて動作する．クラスファイルとは，Java バーチャルマシンの命令列 (バイトコード) とシンボル・テーブル，ならびに他の付随的な情報を保持したものである．バイトコードは，クラスファイルにおいて一定個数の引数を渡すことのできる実行可能なコード (メソッド) として記述されている．

Java バーチャルマシンはセキュリティを考慮して，クラスファイルのコードに厳格なフォーマッ

トと構造上の制約を課している．しかし，有効なクラスファイルを表現するだけの機能を持った言語であればどのようなものでも，Java バーチャルマシンをホストとすることができる．汎用性があり，マシンに依存しないプラットフォームを持つという点から，他の言語実装者もその言語を配信する仕組みとして Java バーチャルマシンに注目している．

さて，Java バーチャルマシンは，同時に複数のスレッドを実行することが可能である．すなわち，共有メモリ中の値を操作する Java バイトコードを，並列に実行することが可能である．

また，実行するメソッドへのポインタ，ならびに参照する変数へのオブジェクト内のオフセットは実行時に決定される．

Java バーチャルマシンは，スタックマシンである．このため，扱うデータ域および命令セットは特徴的なものとなっている．以下にそれらについて述べる．

2.2.1 データ域の特徴

Java バーチャルマシンが実行時に使用するデータ域は，図 1に示すように大きく 5 つの部分から成る．

- プログラムカウンタ
- Java スタック
- ヒープ
- メソッド・エリア
- コンスタント・プール

以下に，それぞれの詳細を述べる．

プログラムカウンタ

Java バーチャルマシンは，複数のスレッドを同時に実行することができる．Java バーチャルマシンの各スレッドには，プログラムカウンタが個別に存在している．プログラムカウンタが保持する値は，実行中のメソッドの性質により異なる．メソッドがバイトコードで記述されたものである場合，プログラムカウンタは現在実行中のメソッドにおけるバイトコードの命令アドレスを保持している．現在実行中のメソッドがネイティブコードで記述されたものである場合，プログラムカウンタの値は未定義である．

Java スタック

Java バーチャルマシンの各スレッドは，スレッドと同時に生成される固有の Java スタックを保持している．Java スタックとは，C 言語が使用するような従来のスタックに相当するもので，ローカル変数と結果の一部を保持し，メソッドの呼び出しおよびリターンを効率良く行うためのものである．なお，Java スタック用のメモリ領域は連続している必要はない．

Java スタックにはフレームが格納される．フレームとは，参照する変数およびメソッドへのオフセット計算，メソッドの値の返却，例外処理の実行，演算結果の一部の格納などに使用されるものである．フレームは，メソッド呼び出し毎に新たに生成される．メソッドの正常終了か異常終了かに関わらず，フレームはメソッドの終了時に破棄される．各フレームは，ローカル変数とオペランド・スタックを個別に保持している．

- ローカル変数

  ワードの一次元配列としてフレーム内に割り当てられ，この配列の基底からのワード・オフセットとしてアドレス付けされる．サイズはコンパイル時に決定される．

  ローカル変数の大きさは必ず 1 ワードである．2 ワードの記憶領域を必要とする型に対しては，2 つのローカル変数領域が確保される．

- オペランド・スタック

  Java バーチャルマシンの大半の命令は，現在実行中のメソッドに対応するフレームのオペランド・スタックから値を取得，演算した後，同一のオペランド・スタックに結果を返す．また，オペランド・スタックは，メソッドに引数を渡し，その結果を受け取るためにも使用される．使用するオペランド・スタックのサイズの最大数はコンパイル時に決定される．

  ローカル変数と同様，オペランド・スタックの各エントリの大きさは 1 ワードである．2

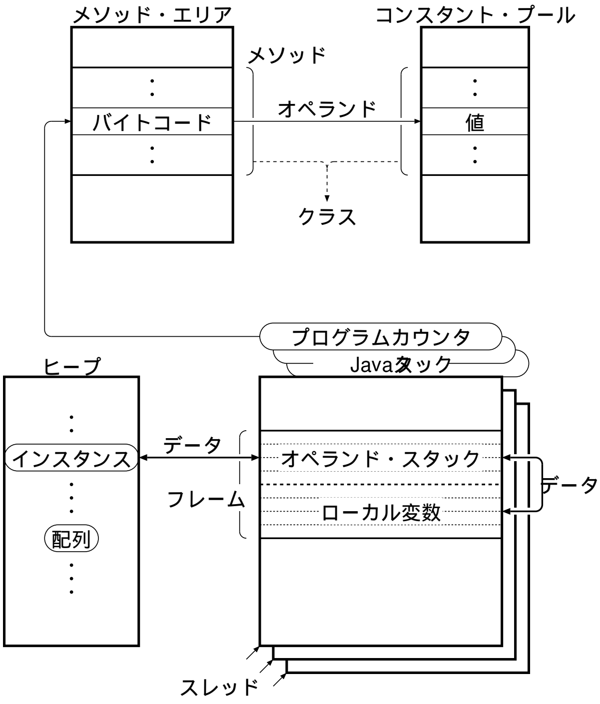


図 1: Java バーチャルマシンのデータ域

ワードの記憶領域を必要とする型に対しては，2 つのエントリが使用される．

スレッドが生成するフレームはそのスレッドだけが参照可能であり，他のスレッドが直接参照することはできない．メソッド呼び出し時において引数を渡す，もしくはリターン時に値を返す処理においてのみ他のフレームとのデータのやりとりが可能である．

ヒープ

Java バーチャルマシンは，すべてのスレッド間で共有されるヒープを保持している．ヒープとは，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データ域である．

Java バーチャルマシンのヒープは，Java バーチャルマシン自身の初期化時に生成される．オブジェクトによって使用されたヒープ領域は，自動記憶管理システムによって管理される．すなわち，オブジェクトは明示的に再割り当てされるわけではない．Java バーチャルマシンは特定の動作をする自動記憶管理システムを想定しているわけではなく，実装者はシステム要求に沿った記憶管理方法を選択することができる．ヒープは，固定サイズとすることもできるし，必要に応じて拡張したり，大きなヒープが不要になった際に縮小することもできる．Java バーチャルマシンのヒープ用メモリ領域は，連続している必要はない．

メソッド・エリア

Java バーチャルマシンは，全てのスレッドで共有されるメソッド・エリアを保持している．メソッド・エリアとは，従来の言語におけるコンパイル済のコードの格納領域，具体的には UNIX プロセスの「テキスト・セグメント」に似たものである．メソッド・エリアは，クラスを単位としてメソッドを格納している．メソッド・エリアは，Java バーチャルマシンの初期化時に生成される．メソッド・エリアは固定サイズとすることもできるし，必要に応じて拡張したり縮小したりすることもできる．

コンスタント・プール

コンスタント・プールとは，クラスファイルに存在するコンスタント・プール・テーブルの実行時におけるデータ構造である．この中には，コンパイル時に決定される数値リテラルから，実行時に

解決されなければならないメソッド，変数へのリファレンスなど数種類の定数が含まれている．コンスタント・プールは，従来のプログラミング言語におけるシンボル・テーブルの機能に類似した役割を果たすものであるが，典型的なシンボル・テーブルよりも広範囲のデータを保持している．

2.2.2 Java バイトコードの特徴

Java バーチャルマシンの命令は，実行するオペレーションを定義した 1 バイトのオペコードの後に，引数を指定する 0 個以上のオペランドと，オペレーションが使用するデータが続く構成となっている．オペランドを持たず，1 バイトのオペコードのみで構成される命令も数多く存在している．

Java バーチャルマシンの命令セットは以下のように分類することができる．

- ロード・ストア

  ロード命令とストア命令は，Java バーチャルマシンのローカル変数とオペランド・スタックの間で値を移動させるものである．また，オペランド・スタックへの定数のロードも行う．

- 演算

  オペランド・スタック上の値を用いて演算を行い，この値をオペランド・スタックに積むものである．演算命令には大きく，整数演算命令と浮動小数点演算命令とがある．

- 型変換

  Java プログラミング言語は強く型付けされているため，型変換命令を使用することによって，Java バーチャルマシンの数値型間の変換を行う必要がある．また，型変換命令は，Java バーチャルマシンの命令セットにおける直交性の欠如を補う役割も果たしている．

- オブジェクトの生成と操作

  クラス・インスタンスおよび配列はともにオブジェクトである．しかし，クラス・インスタンスおよび配列の生成，参照，更新のためには各々異なる命令を用いる．ヒープ領域のメモリ管理は自動的に行われるため，これらの命令が扱うオブジェクトのリファレンスは，実際のヒープのアドレスとは異なる．このため，ヒープを参照するためには，リファレンスからアドレスへの変換が必要となる．

- オペランド・スタックの管理

  オペランド・スタック上のデータに対して交換，複製したり，プッシュ，ポップを行うための命令が定義されている．

- 分岐

  Java バーチャルマシンでは，条件分岐，無条件分岐，テーブルジャンプ，サブルーチンコールのための命令が定義されており，オペランド・スタック上の値の比較および分岐を 1 つの命令で記述することができる．

- メソッドの呼び出しとリターン

  メソッドの呼び出し時には，新しくフレームを生成する．また，フレーム間の引数の受け渡しは，呼び出したメソッドに対応するフレームのオペランド・スタック上の引数を，呼び出されたメソッドに対応するフレームのローカル変数に移動することにより行われる．実行すべきメソッドの先頭アドレスは，メソッドの属するオブジェクトのリファレンス，およびメソッド呼び出し命令のオペランドであるコンスタント・プールへのインデックスを用いて計算される．インスタンスメソッドの呼び出し時には，暗黙の引数としてそのオブジェクトへのリファレンスを渡す．

  メソッドのリターン時には，呼び出されたメソッドに対応するフレームのオペランド・スタック上の戻り値を，呼び出したメソッドに対応するフレームのオペランド・スタックへと移動することにより値の受け渡しを行う．実行の終了したメソッドに対応するフレームは破棄される．

- 例外のスロー

  Java バーチャルマシンの命令セットは，明示的に例外を発生させるための命令を持っている．

- 同期

Java バーチャルマシンは，メソッドおよびブロック・レベルでの同期をサポートしている．同期メソッドは，メソッド呼び出しおよびリターンの一部として取り扱われる．一方，同期ブロックのためには明示的な命令が用意されている．

2.3 Java バイトコードの実行方式

Java バーチャルマシンは，指定したクラスの `main` メソッドに対して，文字列の一次元配列である引数を 1 つ渡し，呼び出すことにより実行を開始する．この時点では，指定されたクラスはロードされていない．このため，Java バーチャルマシンは，クラスのバイナリ表現をまだ保持していないことを判別しロードする．この後，ロードしたクラスが正しいか否かを検証する．検証の結果が正しいと判明すると，次にクラスを保持するための準備を行う．また，このクラスが使用している他のクラスへのリファレンスが正しいかどうかを調べるために，参照先のクラスについてもロード，検証などの一連の処理を行う．これらの処理を終えた後，メソッド内のバイトコードの実行を開始する．メソッドの実行開始後も，新しいクラスを使用する時には，前述した一連の処理を行う．

さて，バイトコードの実行方法には，Java バーチャルマシンの実装方式により，以下の 3 つが考えられる．

1. インタプリタ方式

   プログラムの実行時にバイトコードを 1 つずつ翻訳し，ネイティブコードにより記述されたあらかじめ用意した各バイトコードに対応する関数を呼び出すことにより実行する方法．

2. Just-In-Time(JIT) コンパイラ方式

   バイトコードを 1 つずつ翻訳していくのではなく，いくつかのバイトコードを単位として最適化を行いネイティブコードに変換しながら実行する方法．一度ネイティブコードに変換された命令群はメモリ上に記憶され，再度実行する際にはコンパイルを行う必要がない．

3. ハードウェアによる直接実行

   Java バイトコードを直接実行することが可能であるハードウェア上で実行する方法．

前述のように，Java プログラミング言語のプログラムからバイトコードを生成する Java コンパイラは，コンパイル時の最適化を十分に行うことができない．つまり，Java バーチャルマシンの高速化は，コンパイル時ではなく，バイトコードの実行時に行う必要がある．

JIT コンパイラ方式は，既存のプラットフォーム上でバイトコードを高速に実行する手法であり，Java バイトコードからネイティブコードへの変換の際に最適化を行う．ただし，本方式はプラットフォーム毎に最適化の内容が異なる．

さて，Java バーチャルマシンの高速化手法として，本稿ではデータ再利用の手法を提案する．本手法は，実行結果を保存しておき，Java バーチャルマシンが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．詳細を次節において述べる．

2.4 データ投機とデータ再利用の違い

データ投機とデータ再利用との違いは，投機的であるか否かである [5]．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．このため，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．

命令実行の高速化手法として，データ投機 [6] がある．分岐予測においては分岐命令の結果を予測するのに対して，データ投機の手法はその命令の実行結果を予測する．ある命令の実行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．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場合には，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．データ投機の手法においては予測した値が正しいかどうかの判断のため，予測の対象となった命令を実行する必要がある．予測値を用いて実行する命令に対しても予測を行うためには，予測を行った各時点におけるメモリの状態を保存しなければならない．このため，

予測値とそれ以後更新されたメモリの状態を対応して保存しておく必要がある．

データ再利用の手法は，本来プロセッサ内部における高速化手法として研究されてきた[7]．例えば，浮動小数点数演算は演算ステージに要するサイクル数が長い．このため，以降の命令がその実行結果を必要とするとパイプラインがストールしてしまう．これを解決するために，一度行った演算結果を保存しておき，再度同じ内容のオペランドを用いて演算する場合に演算結果を再利用することでクリティカルパスを短縮する．

この後，浮動小数点数演算以外の命令についても同様に，演算ステージを省略するという研究が行われた[8]．先に述べた手法と同様にオペランドの値が一致した場合に，演算結果をバッファから読み出すことにより演算ステージの実行を省略し高速化する．演算ステージの実行を省略することが可能であれば，演算結果を早く得ることができる上に資源競合の解消にもつながる．

この手法を用いれば，分岐予測が正しくなかったことによるペナルティを軽減することができる．これは，投機的に実行された命令の実行結果のうち分岐命令に依存しないものは，予測が正しくなかった場合その実行結果の再利用が可能であるからである．また，関数内で実行される命令は全ての引数を参照するわけではない．その命令が参照する値さえ一致すれば再利用は可能である．関数の引数の一部の値が一致する場合は非常に多く，この手法は効果的である．

また，単一の命令を対象としたデータ再利用ではなく，複数の命令を単位としたデータ再利用も考えられている[9]．複数の命令をデータ再利用の単位とした場合，普通の命令と同様に入力と出力が存在する．命令群内において生成され，同一命令群内においてのみ使用される値はレジスタへ書き込む必要はない．すなわち，この手法は各命令間にデータ依存が存在する場合に有効である．データ再利用の対象となる命令の単位は，データ再利用が可能ではない命令により区切られた命令群，徐々にその単位を拡張するという2通りの手法が提案されている．データ再利用の手法は，メモリ上の値は常に正しいものであるため，その機構はデータ投機の手法と比較して単純に構成することが可能である．

以上に述べたように，データ投機の手法は，予測値が正しいかどうかの判断を行うため予測の対象となった命令を実行する必要がある．一方，データ再利用の手法は，入力データの値が等しくデータ再利用が可能である場合には，その命令を実行する必要はない．すなわち，データ投機の手法は資源競合が発生する場合において適用することが不可能であるのに対して，データ再利用の手法は実行する処理を省略することが可能であるため，資源競合を解決することが可能である．

Javaバーチャルマシンは各命令間のデータの依存関係が強いため，本稿では，単独の命令ではなく複数の命令を単位としてのデータ再利用を採用する．

3 Value Lookaside Buffer(VLB)

3.1 命令の分類

データ再利用は，全てのJavaバイトコードに対して適用できるわけではない．本節では，複数の命令を単位としてデータ再利用を行う場合に，バイトコードを実行結果の再利用が可能であるものと不可能であるものに分類する．

3.1.1 データ再利用が不可能であるもの

全てのJavaバイトコードの中からデータ再利用が不可能であるものを以下に列挙する．

- new，newarray，anewarray，multianewarray

  ヒープ上にメモリ領域を確保し，クラス・インスタンスもしくは配列の生成を行う命令である．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，データ再利用は不可能である．

- athrow

  例外もしくはエラーをスローする命令である．現在実行中のメソッドに例外ハンドラが存在する場合には，オペランド・スタックをクリアし，例外もしくはエラーを意味するオブジェクトをスタックトップへと積み，ハンドラの

実行を開始する．これらの処理はデータ再利用が可能である．しかし，現在実行中のメソッドにハンドラが存在しない場合には，例外を取り扱うメソッドの実行まで戻り，その間に呼び出されたメソッドに対応するフレームを全て破棄する．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るため，データ再利用は不可能である．

- 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る．

以上に挙げたバイトコードは，実行結果を再利用することが不可能である．

3.1.2 組み合わせによりデータ再利用が可能となるもの

バイトコード単体では再利用が不可能であっても，別のバイトコードと組み合わせることによりデータ再利用が可能となるものが存在する．このような特徴を有するものを以下に列挙する．

- invokevirtual, invokespecial, invokestatic, invokeinterface

  メソッドを呼び出す命令である．これらは，引数が同じである場合にも生成するフレームは異なる．すなわち，実行結果を再利用することは不可能である．

- ireturn, lreturn, freturn, dreturn, areturn, return

  メソッドからリターンする命令である．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処理を行う．オペランド・スタックの状態だけでは，元のフレームを特定できないため，データ再利用は不可能である．

以上のバイトコードは単体ではデータ再利用が不可能である．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，フレーム操作を行う必要がなくデータ再利用が可能となる．

3.1.3 データ再利用が可能であるもの

3.1.1項および3.1.2項で述べた以外の命令は，データ再利用が可能である．メモリアクセス命令における入力データは，そのメモリアドレスだけではなくメモリ上の値をも含み，その実行結果はオペランド・スタックから他のメモリ領域への書き込みである．単一の命令を単位としてデータ再利用を行う場合，再利用することは命令を実行することと等しくなる．これでは，データ再利用を行うことによる高速化は望めない．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必要がある．

3.2 再利用を行う単位

データ再利用の手法を Java バイトコードの実行時に適用することを考える．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どうかを判断する際に，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ことを示している．すなわち，ある命令の実行結果が再利用できるなら，直後の命令の実行結果も再利用できる可能性が高い．よって，本稿で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とする．

本節では，再利用を行う単位について検討を加える．

3.2.1 区切り命令を用いる場合

まず考えられるのは，図 2に示すようにデータ再利用が不可能であるバイトコードを区切りとして，実行時のバイトコードを複数の命令群に分割する方法である．ただし，区切りとなる命令をデータの再利用が不可能である命令のみに限定すると，各命令群が長くなる．その結果，入力データが増加してしまい，以前の実行結果と比較した場合に一致するものの数が少なくなってしまう．これに

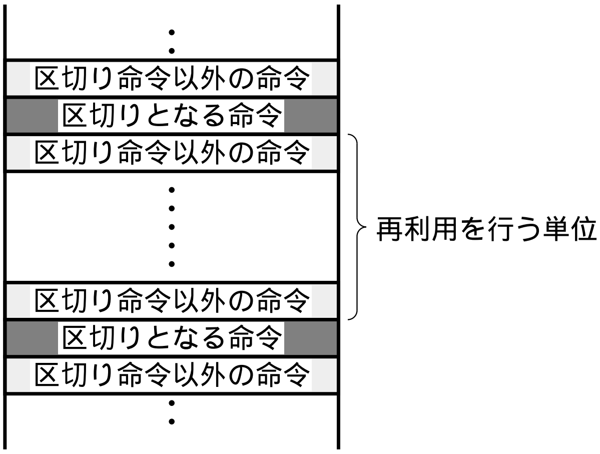


図 2: 区切り命令を用いる場合

より，十分にデータの再利用を行うことは不可能である．そこで以下に，この区切りとなるバイトコードの選択方法について考察を行う．

演算結果はオペランド・スタックのスタックトップに存在する．バイトコードの実行時には，1) 必要な値をオペランド・スタックに積み，スタック上の値を用いて演算を行う; 2) その後この演算結果を，メモリにストアする/条件分岐命令の判断に使用する/メソッド呼び出しの引数として使用する; などの処理を行う．演算結果を使用する命令を用いてデータ再利用の単位を区切ることが，Java バーチャルマシンにおける実行結果の再利用に適していると考えられる．

データ再利用が可能である命令のうちオペランド・スタックの値を取り出して使用する命令には，ローカル変数へのストア命令，ヒープ領域へのストア命令，演算命令，型変換命令，条件分岐命令が挙げられる．しかしこのうち，演算命令，型変換命令，分岐命令およびローカル変数へのストア命令は，各フレーム内におけるデータアクセスである．一方，ヒープへのストア命令は，各スレッド間で共有するデータ領域へのアクセスである．このため，厳密に行う必要がある．そこで，区切りとなる命令として，再利用が不可能な命令にヒープへのストア命令を追加する．以上の方針から再利用を行う単位を区切るための命令を以下のものとした．

- データ再利用が不可能な命令

  new, newarray, anewarray, multianewarray, athrow, monitorenter , monitorexit

- 単体ではデータ再利用が不可能な命令

  invokevirtual, invokespecial, invokestatic, invokeinterface , ireturn, lreturn, freturn, dreturn, areturn, return

- 配列へのストア

  iastore, lastore, fastore, dastore, aastore, bastore, castore , sastore

- スタティック変数へのストア

  putstatic

- インスタンス変数へのストア

  putfield

3.2.2　メソッドを単位とする場合

メソッド呼び出しおよびリターン命令に注目する．3.1.2項で述べた通り，メソッド呼び出しからリターンまでを単位とするデータ再利用を行うことにより，広範囲の再利用が可能となる．この方式においては，メソッド内に存在するバイトコードの実行の省略に加え，メソッド呼び出し時のオーバヘッドも削減することが可能となる．そこで以下に，メソッドの実行を単位としてデータ再利用を行うことが可能であるかどうかを述べる．

Java バーチャルマシン上の各スレッドにおいて，メソッド呼び出しは図3に示すように入れ子状に行

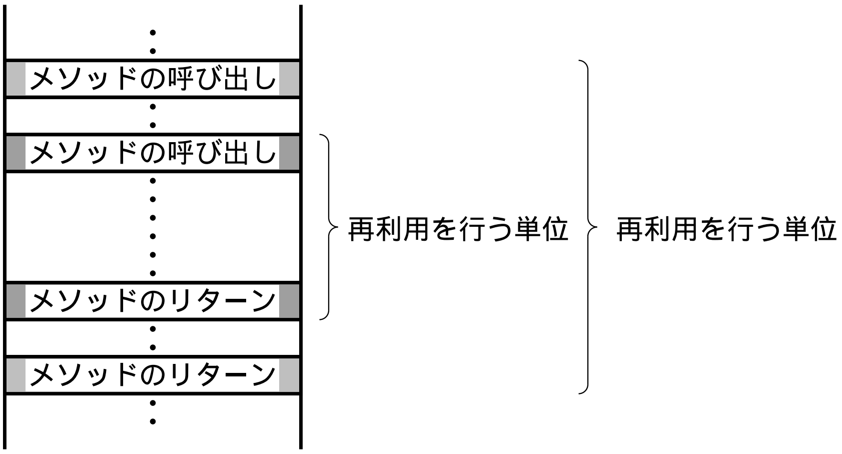


図 3: メソッドの実行を再利用の単位とする場合

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．データ再利用が可能であるかどうかの判断は，命令実行に必要となった入力データを以前のものと比較することにより行われる．内側のメソッドの実行時に必要となった入力データは，外側のメソッドの実行時にも必要となった入力データとなる．実行結果についても同様のことが言える．メソッドの呼び出しがどれだけ入れ子になったとしても，このメソッドの実行中に参照した入力データおよび実行結果さえ保存すれば，データ再利用を行うことが可能である．またデータ再利用を効率良く行うため，複数のスレッドにまたがって実行する場合においても，実行結果を登録する領域は全スレッドで共有する．

ただしこの場合には，3.1.1項で述べた，データ再利用が不可能な命令を実行する場合が存在する．データ再利用が不可能な命令を実行した場合には，現在のスレッド内で実行中の全メソッドが再利用できなくなる．また，ネイティブメソッドを実行した場合にも，現在のスレッド内で実行中の全メソッドは再利用できない．

3.3 VLB の概要

Java バイトコードが扱うメモリ領域は，オペランド・スタック，ローカル変数およびヒープである．データ再利用が可能であるかどうかの判断は，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，これらのメモリ領域に対して実行結果を書き込む．データ再利用を行うためには，これら入力データおよび実行結果を保存しておくためのメモリが必要となる．このメモリを Value Lookaside Buffer(**VLB**) と呼ぶ．

具体的には，VLB には図 4のような情報が保存される．

入力データ

- データ再利用の開始地点

  これは，データ再利用の単位である命令群を識別するために保持するものである．開始地点とは，区切り命令を用いる場合においては，区切り命令の存在するプログラム上の位置，メソッドを単位とする場合においては，メソッドの先頭である．

  プログラムカウンタは，現在実行しているバイトコードのメソッド内での位置を保持するものであるため，区切り命令を用いる場合のデータ再利用の開始地点を保存するために，プログラムカウンタの他に実行しているメソッドの情報も保存する．メソッドを単位とする場合には，メソッドの情報のみを保持する．

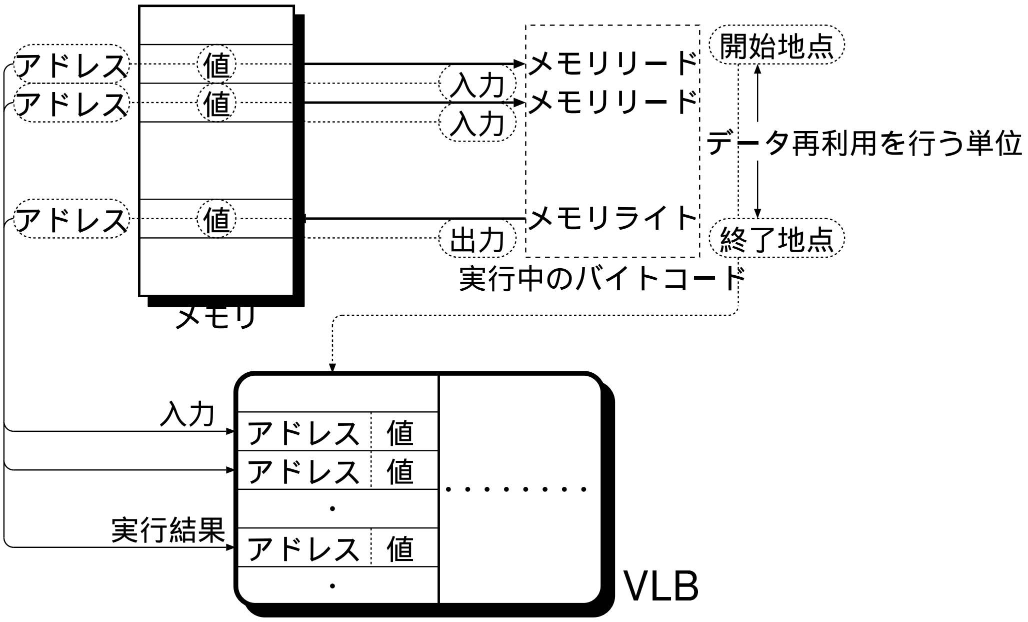


図 4: データ再利用に必要であるデータ

- データ再利用を行う単位に到達した時点におけるメモリの状態

  データ再利用の単位である命令群が参照したメモリ領域の値を保存する．保存する値は参照したメモリアドレスおよびそのアドレスに存在する値の組である．対象となるメモリ領域は，区切り命令を用いる場合には，オペランド・スタック，ローカル変数およびヒープである．一方，メソッドを単位とする場合には，オペランド・スタック上の引数およびヒープである．

実行結果

- データ再利用を行う単位の終了地点

  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を VLB に保存する．メソッドを単位とする場合は，メソッド呼び出し命令の次にプログラムに記述された命令を実行するため，命令アドレスを保存する必要がない．

- データ再利用を行う単位の命令群の実行を完了した時点におけるメモリの状態

  データ再利用の単位である命令群の実行中に更新されたメモリの値を保存する．入力データの場合と同様に，メモリアドレスおよびその値を組にして保存する．対象となるメモリ領域は，区切りを命令用いる場合は，オペランド・スタック，ローカル変数およびヒープである．一方，メソッドを単位とする場合は，メソッドの戻り値およびヒープである．

Java バイトコードの実行時にこれらの情報が VLB に登録される．データ再利用が可能であるかの判断は，区切り命令を用いる場合には区切り命令の実行直後，またメソッドを単位とする場合にはメソッド呼び出しの直前の時点にて行われる．この時に行われる処理は，現在のメモリの状態と VLB に登録されている時点におけるメモリの状態の比較である．全ての値が同じであれば，登録されている実行結果をメモリに書き戻し，実行を省略した次の命令の実行を開始する．比較の結果が異なれば，データ再利用が不可能である．この時は，データ再利用が可能かどうかの判断の対象となった命令群の実行を開始し，実行結果を VLB に登録する．

メモリリードのアドレスおよび値の VLB への

登録は，どのアドレスの値を保存すれば良いのかがわからないため，データ再利用の開始地点で行うことは不可能である．そこで，メモリリードが発生した時点において，VLB への登録を行う．この時，メモリリードにより読み出した値が同じ命令群で書き込まれたものである場合，このメモリリードは比較の対象とはしない．これは，このリードにおいて読み込まれた値が，開始地点におけるメモリの状態ではないからである．また同じ理由により，同一アドレスに重複してリードを行う場合には，最初に行われたリード時の値を有効とする．

3.4 VLB の構成

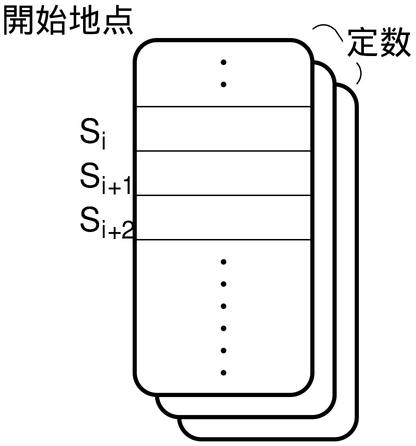


図 5: VLB の構成の例

VLB の構成を登録の観点から考える．まず，データ再利用の開始地点をインデックスとして VLB へ登録する方法を考えてみる．この場合の構成を図 5に示す．本構成では，VLB に登録することが可能な命令群の種類が各開始地点で等しくなる．例えば開始地点の等しいものを同時に登録することが可能な VLB のエントリ数が 10 であるとする．S_iから開始される命令群が 20 通りに実行されるとする．また S_{i+1}から開始される命令群は必ず実行結果が同じであるとする．この場合，S_{i+1}から開始される命令群はエントリを 1 つしか使用せず，残りの 9 個のエントリは未使用となる．一方，S_iから開始される命令群は 20 通りに実行されるが，そのうち 10 通りしか VLB に登録することができない．このため，VLB 上に以前の結果が存在する確率は半分しかない．すなわち，領域割り当ての不均衡は，効率の良いデータ再利用の妨げとなる．S_{i+1}から開始される命令群の実行結果を登録しなかった 9 個のエントリを，S_iから開始される命令群で使用することができると仮定すれば良い．この場合，20 通りに実行される S_iから開始される命令群のうち 19 通りを VLB に登録することが可能となる．

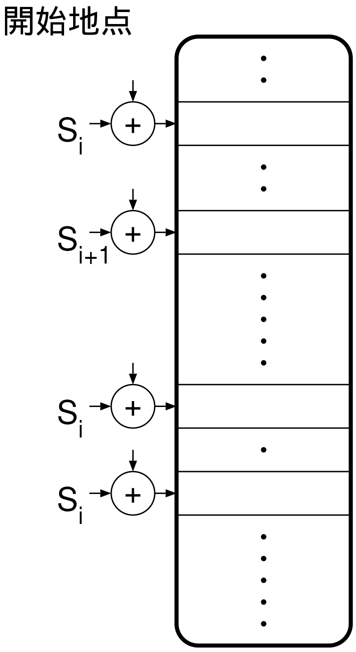
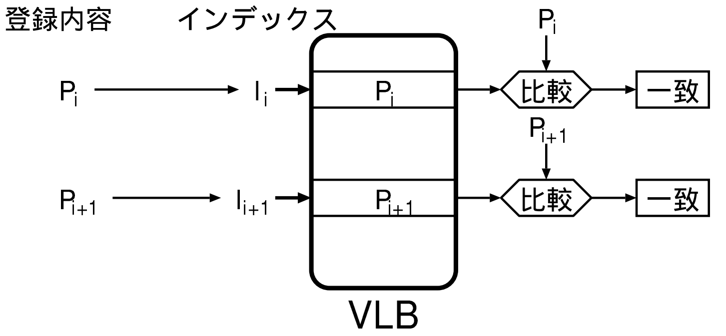


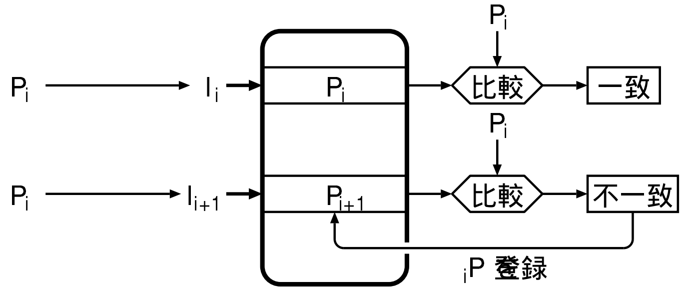
図 6: VLB の構成

そこで VLB へのインデックスを計算するために，登録開始地点の情報以外の情報を用いることとした．この結果，図 6に示すように，実行回数の多い命令群は VLB に多数登録される．また，実行回数の少ない命令が使用する VLB のエントリ数は実行回数に比例して少なくなる．すなわち，VLB の領域を各命令群の実行回数により均等に使用することが可能となる．

一方，登録が可能であっても，再度実行する場合に同一エントリへのアクセスを行わなければ，実行結果を再利用することは不可能である．図 7に示すように，VLB と比較する際に参照するアドレス上の値と異なる値を用いてインデックスを計算すれば，実行する毎にそのインデックスの値は異なる．そこで，データ再利用が可能であるかどう



登録内容　よりVLBのインデックスが一意に　定する



登録内容によりVLBの　ンデックスが一意に　定しない

図 7: インデックスと再利用の可能性の関係

かの判断に使用するメモリの値を用いて，VLBへのインデックスを求める．しかしVLBにアクセスを行わない限り，どの値を使用して比較を行うのかはわからない．そこで，どのメモリ上の値を使用して，VLBへのインデックスを求めると，図7の下に示すような例が起こりにくいのかについて考える．3.2節で述べたように，登録する値が異なるため，各場合について以下に述べる．

- 区切り命令を用いる場合

  区切り命令を用いる場合には，オペランド・スタック，ローカル変数およびヒープの値が登録される．区切りとなる命令を決定した経緯から考えると，この命令群で行われる処理に必要な入力データは，この命令群の実行が開始されてからオペランド・スタックに積まれる．つまりオペランド・スタック以外のメモリ領域に，データ再利用が可能かどうかの比較の対象となる値が存在する．そこで，アドレスの特定の困難なヒープではなく，アクセスの容易なローカル変数領域に存在する値を使用することにする．

- メソッドを単位とする場合

  この方式の場合には，オペランド・スタック上の引数およびヒープ上の値をデータ再利用が可能かどうかの判断の比較に用いる．引数はそのメソッドで使用するために渡されるものである．また，ヒープと比較してオペランド・スタックへのアクセスは容易である．このため，引数を使用してVLBへのインデックスを求める．

3.5 VLBを用いたデータ再利用

前節までに述べた方針に従ってVLBを実装した．データ再利用が行われる手順について例を用いて詳細に述べる．

図8に示すメソッドaddは，インスタンス変数xに引数yを加算し，その結果を戻り値として返す．Javaバーチャルマシンは，引数をローカル変数として受け取る．このため，メソッドaddの属

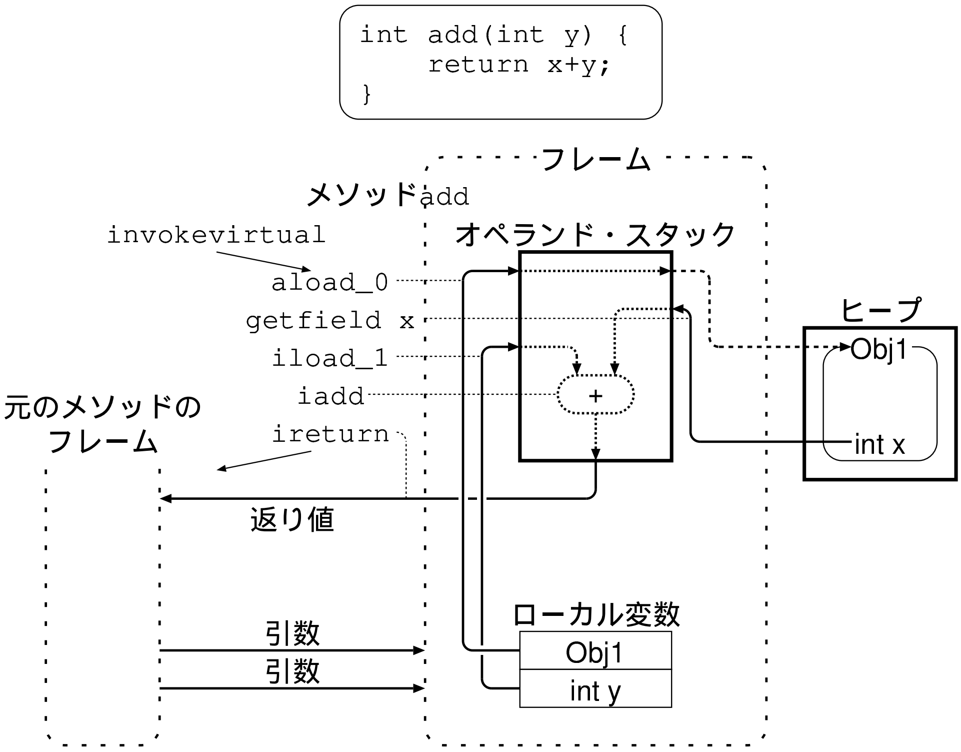


図 8: バイトコード実行の例

するオブジェクト (Obj1) へのリファレンス，および整数値 y の 2 つの引数は，メソッド`add` に対応するフレームのローカル変数領域に格納される．

メソッド`add` は呼び出されると，まず`aload_0` の実行により 0 番のローカル変数の値をオペランド・スタックへと積む．次に，`getfield` の実行により先ほど読み込んだオブジェクトのリファレンス，および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から計算されたアドレスを用いてヒープ領域のリードを行う．その実行結果であるインスタンス変数 x の値をオペランド・スタックへと積む．その後，`iload_1` の実行により 1 番のローカル変数の値をオペランド・スタックへと積み，`iadd` の実行により先程のインスタンス変数 x の値との加算の結果をオペランド・スタックへと返す．`ireturn` の実行により，その値を戻り値として元のフレームへと渡す．

以下に，この例におけるデータ再利用の動作を，区切り命令を用いる場合，およびメソッドを単位とする場合，それぞれについて述べる．

3.5.1 区切り命令を用いる場合

この例において区切りとなる命令は，メソッド呼び出し命令である`invokevirtual`，およびメソッドリターン命令である`ireturn` である．この 2 つの命令間に存在する 4 つの命令がデータ再利用の単位である．

この命令群が初めて実行された時点では，VLB にはこの命令群の実行結果に関する情報は，何も登録されていない．このため，VLB のエントリには比較の対象となるものが存在しない．そこで，この命令群は実行され，実行結果が VLB へと登録される．この登録の様子を，図 9に示す．VLB へのインデックスは，このメソッドの情報，`aload_0` の命令アドレス，`aload_0` を実行する時点におけるローカル変数 Obj1 および y の値を用いて決定される．

まず，開始地点 (a) および開始地点におけるオペランド・スタックの状態が登録される (b)．この例における登録の開始地点はメソッドが呼び出された直後であるため，オペランド・スタックは空であ

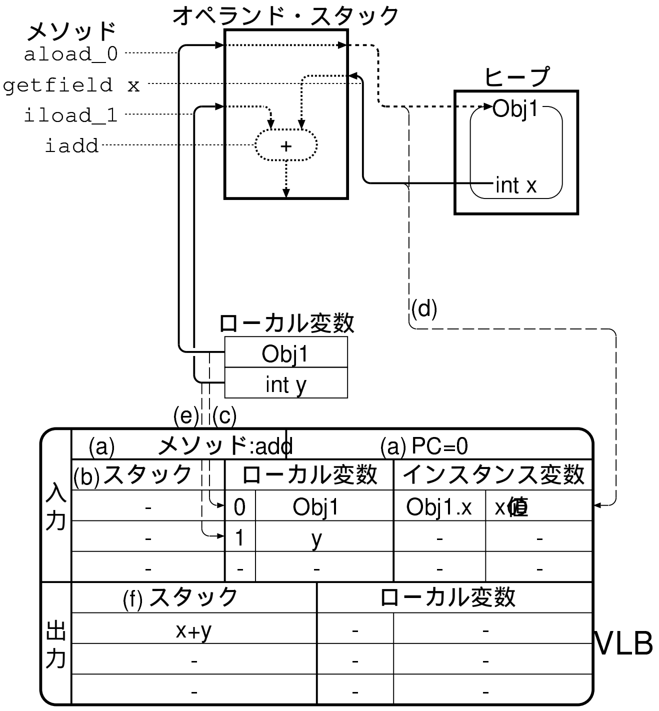


図 9: 区切り命令を用いる場合

る．次に，`aload_0` が実行される．この命令はローカル変数を参照するため，この情報が入力データとして VLB へと登録される (c)．登録されるデータは，ローカル変数番号およびその値である Obj1 へのリファレンスである．次に，`getfield` が実行され，オペランド・スタック上の Obj1 オブジェクトのリファレンス，フィールド x に対応するコンスタント・プールへのインデックス，および読み出した x の値が VLB へと登録される (d)．`iload_1` の実行時に登録されるデータは，`aload_0` と同様である (e)．`iadd` はオペランド・スタック上における値の操作であるため，何も登録しない．

ここまでで，データ再利用の単位となる命令群の実行が完了した．完了時点におけるオペランド・スタックは，命令群の実行結果を含むため，オペランド・スタックの状態である $x+y$ の値も，実行結果として VLB に登録する (f)．以上により，VLB への登録が完了する．

再びこの命令群が実行された時には，VLB にこの命令群の実行結果が存在する．このため，VLB へのインデックスが以前実行した時と一致すれば，データ再利用が可能かどうかの判断が開始される．この場合，`invokevirtual` 命令の実行が完了した時点で，VLB に登録されているメモリの状態と現在のメモリの状態の比較を行う．この例では，以前に登録を行った 0 番のローカル変数の値，1 番のローカル変数の値，および Obj1.x の値の比較を行う．全ての値が一致すれば，データ再利用が可能である．この場合には，VLB に登録されている実行結果をメモリへ書き込む．この例では，ローカル変数の更新は行われないため，オペランド・スタックの状態である $x+y$ の値のみを書き戻す．データ再利用が不可能である場合は，この命令群の実行を開始し，その結果を VLB へ登録する．

また，同時に比較の対象とする VLB のエントリは，1 つではなく複数とする．比較の対象となるエントリと登録の対象となるエントリは同じものである．このため，Obj1.x の値のみが異なるといった実行結果を，それぞれ VLB に登録することが可能である．登録の対象とするエントリに空き領域が存在しない場合には，ランダムに 1 つのエントリを選択しその実行結果を上書きすること

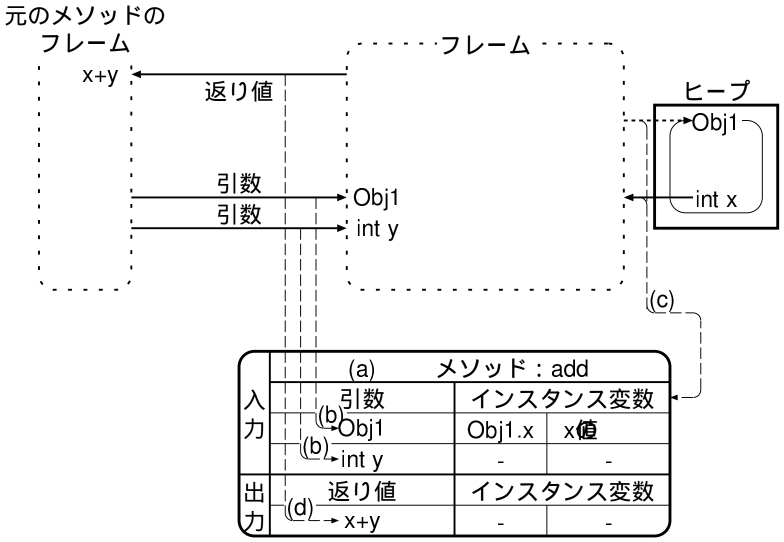


図 10: メソッドの実行を単位としてデータ再利用を行う場合

により登録を行う．

3.5.2　メソッドを単位とする場合

メソッドを単位とする場合は，このメソッドの呼び出しからリターンまでがデータ再利用の単位である．VLB へのインデックスは，呼び出すメソッド，引数 Obj1 および y の値を用いて計算される．

メソッドを単位とする場合においても，初めて実行された命令群は，比較の対象となるエントリが見つからないため必ず実行され，結果を VLB へ登録する．この時の様子を図 10に示す．まずメソッドの実行時に，そのメソッド (a) および引数を VLB に登録する (b)．メソッドの実行を単位とする場合には，フレームの生成を行わない．すなわち，フレーム内でのメモリアクセスは登録する必要はない．`aload_0`，`iload_1` および`iadd` の実行時は VLB への登録は行わない．その後，`getfield` の実行時のヒープのリードが発生すると，ヒープのアドレスおよび値を VLB に登録する (c)．最後に，メソッドが戻り値を持つ場合には，メソッドの実行が完了した時点において，戻り値を VLB に登録する (d)．図の例の場合は戻り値が存在するため，その値 x+y を VLB に登録する．

この時点以降，登録した実行結果を再利用することが可能となる．同一メソッドの実行時に VLB へのインデックスが等しいと，メソッドの入力データの比較を開始する．この例の場合，比較の対象となる値は，メソッドの引数 Obj1，y の値およびインスタンス変数 Obj1.x の値である．これらの値が現在の値と VLB に登録した値とで全て一致した場合，データ再利用が可能である．この時は，メソッドの呼び出しに伴うフレームの生成は行わない．戻り値が存在する場合は元のフレームのオペランド・スタックに戻り値を積む．また，メソッドの実行中に発生したヒープへのライトを行う．この例においては，ヒープへのライトは発生しないため，戻り値 x+y をオペランド・スタックに積む操作のみを行う．データ再利用が不可能である場合は，メソッドの実行を開始しその結果を VLB に登録する．

区切り命令を用いる場合と同様に，比較の候補となる VLB のエントリは複数とする．

3.6　問題点

インタプリタ方式により実装された Java バーチャルマシンに，VLB の実装を行った．本節では，この実装における問題点および解決法について述べる．

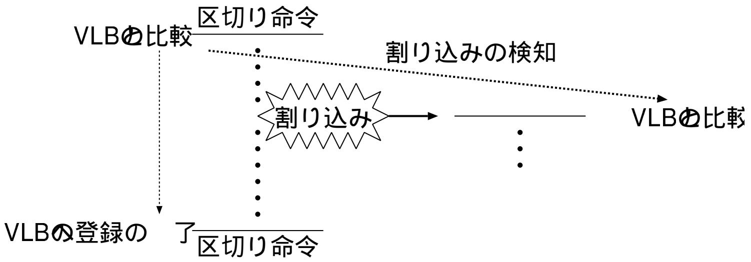


図 11: 割り込みの発生

3.6.1 割り込み

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い．このため，割り込みが発生した時点で登録中の VLB のエントリはデータ再利用に使用することができない．これらのエントリは，登録を行ってはいけない．VLB において，割り込みの発生を検出するための方法を以下に述べる．

- 区切り命令を用いる場合

  区切り命令を用いる場合，メソッドの呼び出しが完了した時点は，データ再利用が可能かどうかの判断を行う時点である．割り込みが発生すると，割り込みハンドラに対応したメソッドの実行が開始される．VLB への登録中に割り込みが発生すると，図 11に示すように，登録が完了する前に別の VLB のエントリとの比較を開始してしまう．このような状態になることは割り込み以外においてはありえない．つまり，このような状態を VLB 側で検知することにより，割り込みが発生したことを知ることが可能であり，現在登録中の VLB のエントリの登録を中止する．

- メソッドを単位とする場合

  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 VLB のエントリの登録を中止する．

3.6.2 実行時のオーバヘッド

データ再利用は Java バイトコードの実行とともに行われる．このため，インタプリタ方式により実装された Java バーチャルマシンに VLB を実装する場合，オーバヘッドが大きいことが予想された．実際，VLB の処理に要する時間は，データ再利用により省略された時間よりも大きいため，インタプリタ方式では高速化に至らなかった．データ再利用による高速化を実現するためには，VLB に対する操作のオーバヘッドを軽減する必要がある．オーバヘッドをバイトコード実行に隠蔽する方法として以下の 2 つの方法が考えられる．

- 並列化

  自動メモリ管理処理はバイトコードの実行とは直接関係しないため，バイトコードを実行するプロセッサとは異なるプロセッサ上で実行することにより並列化することが考えられる．同様に，VLB に関する処理をバイトコードの実行を担当するプロセッサとは別のプロセッサにおいて実行することにより，オーバヘッドをバイトコードの実行時間に隠蔽することができると考えている．

- ハードウェア化

  バイトコードをハードウェアにより直接実行する Java バーチャルマシンに，VLB のためのアシスト機構を付加することにより，バイトコードの実行と，VLB に関する処理を並列に実行することが考えられる．

本稿においては，上記の 2 つの方式のうち，後者の方法について詳細に検討した．次章においてこの方法について詳しく述べる．

表 1: 命令の出現頻度 (単位は%)

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

4 VLB のハードウェア化

4.1 対象とする Java プロセッサ

対象とする Java プロセッサは，データ再利用以外の点からも高速化可能な構成を仮定している．ここで用いられる高速化手法は，実行時における Java バイトコードの命令出現頻度に基づいている．本節では，この Java プロセッサにおいて想定している高速化手法，および Java プロセッサの構成について述べる．

表 1に，SPEC JVM 98[10] の実行時の命令の出現頻度を示す．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．メソッド呼び出し命令は実行するメソッドを決定する操作，ヒープの読み出し命令は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計算が行われ，一般的に多くの処理時間を必要とする．

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

1. オブジェクト変換表 オブジェクトリファレンスを検索キーとし，各オブジェクトの先頭アドレスおよび属性を保持する．
2. フィールド変換表 コンスタント・プールへのインデックスを検索キーとし，各フィールドのオフセットを保持する．
3. メソッド変換表 コンスタント・プールへのインデックスを検索キーとし，各メソッドの先頭アドレスおよび属性を保持する．

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

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る．これにより，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ない．そこで，レジスタに格納できないものは，主記憶に保持する．主記憶上のローカル変数の値を使用して演算を行う場合には，これらの値を一度保持しておくレジスタが必要となる．この場合にはオペランド・スタックのスタックレジスタを介して実行を行う．スタックレジスタの数は，多くとも 1 命令が 1 度に使うスタックのエントリ数だけあれば良い．また，ローカル変数の主記憶アクセスの高速化にはローカルキャッシュを用意する．同様に，オペランド・スタックに対しても高速化のためスタックキャッシュを用意する．

上に述べた点を考慮に入れて仮定したプロセッサ構成を図 12に示す．

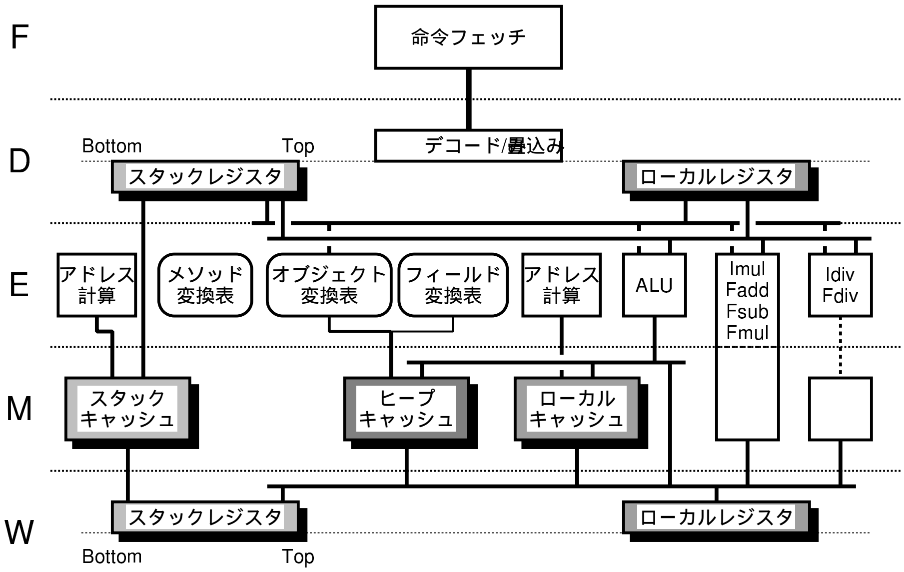


図 12: Java プロセッサの構成

このプロセッサのパイプラインはF/D/E/M/Wの5ステージから構成されている．ローカル変数のうちローカル変数番号の小さいものから一定個数は，レジスタ上に保持することによってオペランド・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う．この値は，一旦スタックレジスタ上に格納された後，演算に用いられる．スタックレジスタはDステージにおいて読み出す．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，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．この処理は必要に応じて自動的にプロセッサ内部で処理されると仮定した．前述の3つの変換表はEステージ，キャッシュはMステージにおいて各々参照される．

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

演算器は，乗算および浮動小数点数の加減算については，データ依存がある場合に2サイクルを要するパイプライン動作と仮定した．また，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．

例えば，図12の構成において，

1. ローカル変数を読み出し，スタックへ積む
2. ローカル変数を読み出し，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命令で扱う全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，これらの命令列は畳み込んで実行する．1.2.の2命令で行われるローカル

変数の読み出しは，ローカルレジスタから D ステージにおいて行う．3. の命令で行われる加算は，先ほど読み出した値を使用して E ステージにおいて行われる．4. の命令において実行されるローカル変数への書き込みは，W ステージにおいてローカルレジスタへと行う．この例の場合，これら 4 命令をプロセッサ内部では 1 命令として処理することが可能である．

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とくらべて，オブジェクト変換表は極めて多くのエントリを必要とする．これは，メソッドおよびクラスは静的に数が決定されるのに対し，オブジェクトは動的に数が決定されるからである．このことから，フィールドおよびメソッドのオフセットを求めるには，プロセッサ上に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．このため，変換対が変換表にない場合のレイテンシを考慮に入れる必要がある．

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．このため局所性が非常に高い．これらのキャッシュはエントリ数の少ないものでもヒット率は極めて高い．一方，ヒープキャッシュは，実現可能なエントリ数の範囲においてはミスヒット時のレイテンシを無視することは不可能である．

以上に述べたことから，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，

- 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
- 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
- M ステージにおいて生成した値を，次命令の E ステージにおいて使用する場合
- メソッドの呼び出し，および，リターン
- 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．

4.2 VLB の実装

4.2.1 データ再利用可能判定処理の軽減

データ再利用の手法は，実行結果を使用する時点よりも前に再利用が可能かどうかの判断を行う．つまり，データ再利用が可能かどうかの判断は，その後にメモリの内容が更新されない限り早く開始することが可能である．データ再利用の手法をバイトコードの実行と並行して行う場合，その判断を早く開始することが可能であれば，その処理に要する処理時間を見かけ上軽減することが可能である．以下に，区切り命令間の命令群を対象とする場合とメソッドの実行を単位とする場合のそれぞれについて，VLB との比較を高速に行う手法を述べる．

区切り命令を用いる場合

データ再利用が可能かどうかの判断を行う地点は，区切りとなる命令の実行直後である．つまり図 12の構成においては，F ステージでフェッチした命令中に区切りとなる命令が存在する場合，VLB との比較をあらかじめ開始しておくことが可能である．

VLB へのインデックスは，メソッド，プログラムカウンタおよびローカル変数を用いて計算する．ローカル変数領域の値はデータ再利用の開始地点に到達するまでに変更される可能性がある．しかし，データ再利用が可能であるかどうかの判断を行う時点からデータ再利用の単位となる命令群の開始時点までに実行するバイトコード数が少ない場合には，ローカル変数上の値はほぼ更新されない．実行結果の再利用が可能であるにもかかわらず VLB へのインデックスが正しくない場合は，実行結果を間違って再利用するわけではない．図 7 に示すように，VLB に二重に登録されるだけである．

区切りとなる命令の実行に先立ち，上に述べた方法を用いて VLB へのアクセスを行う．その結果，比較の対象となるメモリアドレスが求められる．ローカル変数およびオペランド・スタックはレジスタ上に存在するため高速にアクセスすることが可能である．そこで，ヒープ領域の値との比較のみをあらかじめ開始する．バイトコードの実行時にヒープへのアクセスを行おうとする場合，VLB

との比較のためにヒープへのリードを行うと，パイプラインがストールしてしまう．これを避けるため，VLB との比較のためのリードよりもバイトコードの実行時のヒープアクセスを優先して行う．また，あらかじめ比較を行ったヒープの値が変更された場合には，その比較の結果を無効化しなければならない．

またヒープキャッシュを，インスタンス変数，スタティック変数，配列と別々に用意すれば，それらを同時にアクセスすることが可能である．こうすることにより，VLB との比較のためのリードを多数処理することが可能となる．

メソッドを単位とするものの場合

データ再利用が可能かどうか判断を行う地点は，メソッド呼び出し命令の実行の直前である．このため，F ステージでフェッチした命令がメソッド呼び出し命令を含むならば，VLB に登録された内容との比較を，フェッチした時点より開始することが可能である．

VLB へのインデックスは，メソッドおよびメソッドの引数の値を用いて計算される．しかし実行するメソッドは，メソッド呼び出し時にオブジェクトのリファレンスおよびコンスタント・プールへのインデックスを用いて決定される．このうち，オブジェクトのリファレンスはオペランド・スタック上に存在するため，メソッド呼び出し命令を実行するまではどのエントリが実行するオブジェクトのリファレンスであるかを知ることは不可能である．

メソッドの呼び出しに先立って VLB へのアクセスを行うためには，VLB へのインデックスを別の方法を用いて求める必要がある．このため，オブジェクトのリファレンスを用いることなく VLB へのインデックスを求める．この場合メソッドの引数もオペランド・スタック上に存在するため，引数を VLB へのインデックスを求めるために使用することは不可能である．コンスタント・プールへのインデックスのみが，VLB へのインデックスを計算するために使用することが可能である．しかし，メソッドの引数のうち，ローカル変数上からオペランド・スタックへと積まれるものは，ローカル変数上の値を直接 VLB へのインデックスの計算に用いることができれば使用することが可能である．そこで，メソッドの引数のうちローカル変数からオペランド・スタックに積まれるものは，そのローカル変数番号を保存しておく．この値は，コンスタント・プールへのインデックスのみからアクセスすることができる領域に保存する．VLB へのインデックスを求める際には，この値が示すローカル変数上の値を使用する．VLB へのインデックスはデータ再利用が可能であるかどうかの判断に直接使用するわけではないため，図 7に示すように，VLB に同じ内容のものが登録されるだけである．

4.2.2 ハードウェアにおける動作

図 12の構成において，バイトコードを実行する様子を図 13に示す．以下，ハードウェア化した VLB によるデータ再利用の手順について述べる．

データ再利用の開始地点以前の命令実行とオーバラップしてメモリアクセスを行うためには，あらかじめフェッチしておく命令数を十分にとる必要がある．以下では 4 命令をフェッチすると仮定する．フェッチ命令数を増やせば，比較の開始を早めることが可能である．しかし，メソッドの境界を越えては比較は開始できない．また，ヒープの値が更新される可能性が高くなる．データ再利用に有効な VLB へのインデックスを計算することも困難となる．

まず，フェッチした命令中に，データ再利用の開始地点となる命令が存在するかどうかを調べる．図 13の (a) において，データ再利用の開始地点となる命令がフェッチされたとする．

F ステージにおいてフェッチしたバイトコードの中に，データ再利用の開始となる命令が存在した場合，次のサイクル (b) において，先に述べた方法を用いて VLB のインデックス計算を行う．

次のサイクル (c) において，(b) で計算した VLB へのインデックスを用いた VLB の参照を行う．エントリの中に今回実行する命令群と同じ命令群についての実行結果が存在する場合，VLB に登録された情報の中から比較の対象となるメモリアドレスを読み出す．

図 12の構成において，実際にキャッシュアクセスを行うのは E ステージと M ステージである．このため，(d) から (f) までのサイクルは VLB との

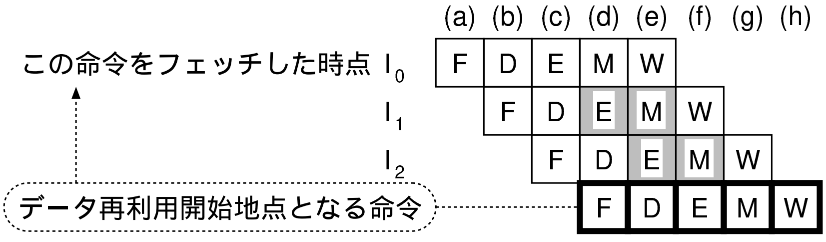


図 13: ハードウェア上でのデータ再利用

比較のためのメモリアクセスが可能である．つまり，命令 I_1 および I_2 がヒープアクセスを行わないならば，その命令の実行と並行して VLB との比較のためのヒープアクセスを行うことが可能である．Java バーチャルマシンはスタックマシンである．このため，オペランド・スタックへのローカル変数のロードは頻繁に発生する．これより，実行する命令がヒープアクセスを行わない可能性は比較的高い．

以降の処理は，区切り命令を用いる場合，メソッドを単位とする場合，それぞれの場合において異なる．以下に，各場合について述べる．

区切り命令を用いる場合

この場合，データ再利用が可能であるかどうかを判断するために必要となるデータは，区切り命令の M ステージを実行した直後の (g) において揃う．これ以降のサイクルにおいて，残りのヒープの値，オペランド・スタックおよびローカル変数との比較を行う．レジスタ上に存在する値は，1 サイクルで全ての比較が可能であると仮定する．等しいエントリが存在しない場合はデータ再利用が不可能である．この場合，図 14に示すように，区切り命令以降の命令を継続して実行する．

データ再利用が可能である場合は，以降の命令の実行を省略する．この場合に次に実行される命令は，VLB により指示されている．この命令がフェッチされている場合は，この命令の D ステージを実行する．そうでない場合，この命令の実行を F ステージから開始する．またサイクル (i) において，実行結果をメモリへ書き込むとともに，次命令の実行に対し，必要となるデータをフォワードする．

サイクル (i) までにデータ再利用が可能であるかどうかの判断が終了し，データ再利用が可能であったとしても，3 サイクルないしは 4 サイクルのパイプラインハザードが生じる．サイクル (i) までにデータ再利用が可能であるかどうかの判断が終了しない場合は，さらに長いパイプラインハザードが生じる．

メソッドを単位とする場合

この場合，メソッド呼び出し命令の直前の命令の M ステージが完了した時点 (g) において，メソッド呼び出し命令の引数が確定する．このため，引数の比較は，次のサイクル (h) において行う．比較の結果が全て等しく，データ再利用が可能である場合，サイクル (i) において戻り値をスタックレジスタに書き込む．また，メソッド呼び出しの直後の命令がこのメソッドの戻り値を使用する場合，VLB に登録された値を次命令の E ステージにフォワードする．

メソッド呼び出しにはパイプラインハザードが伴う．このため，データ再利用が可能かどうかの判断は，メソッドの呼び出しの処理とオーバラップして実行することが可能である．データ再利用が可能であるかどうかの比較が長引けば，その時間だけデータ再利用が可能であった場合に省略できるサイクル数は少なくなる．しかし，データ再利用が不可能である場合と比較すれば，十分効果はある．また比較に必要となった時間の長さに関わらず，メソッドリターン時のパイプラインハザードは削減可能であり，効果的である．

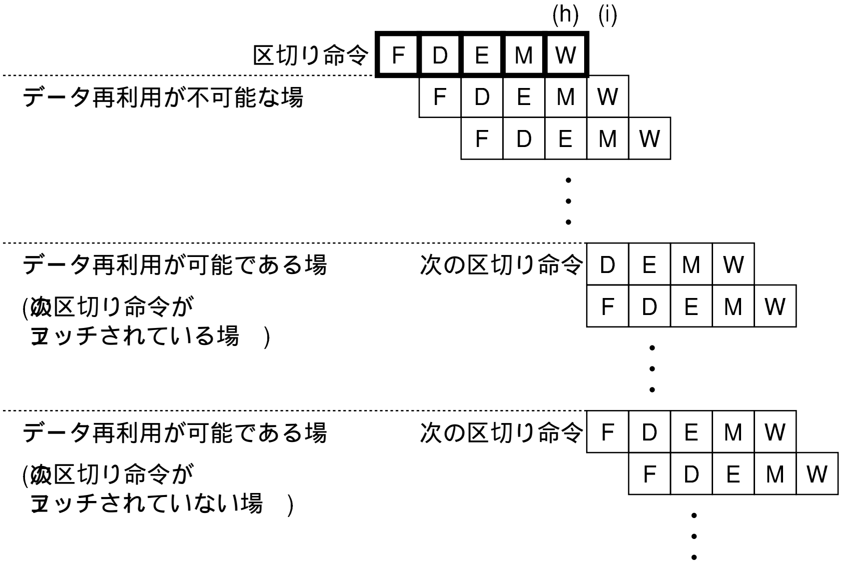


図 14: ハードウェア化した VLB における区切り命令を用いたデータ再利用

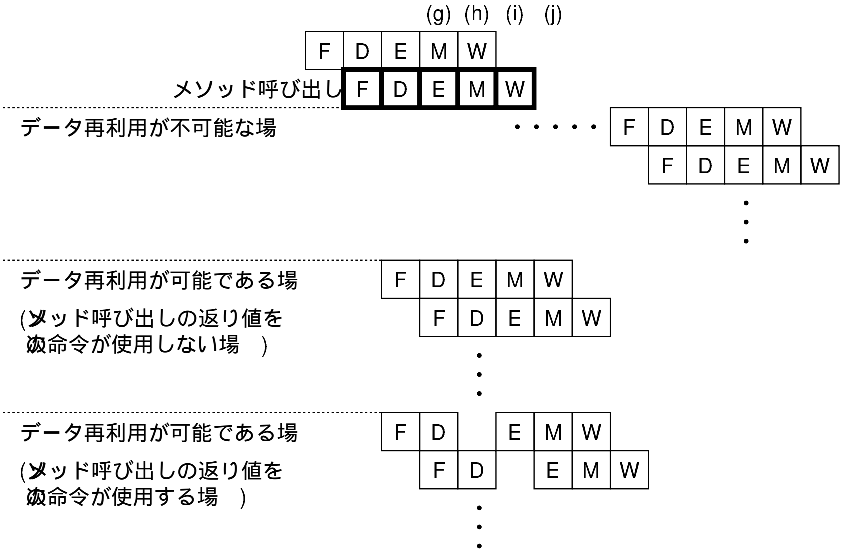


図 15: ハードウェア化した VLB におけるメソッド単位のデータ再利用

5 性能評価

5.1 性能の測定条件

VLB の評価には，SPEC JVM98 VERSION 1.03[10] 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成る．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．しかし，単純に s10,s100 は s1 の 10 倍，100 倍ではない．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

  compress は LZW 法を用いてデータの圧縮を行うものである．SPEC JVM 98 の compress は，SPEC CPU 95 の compress を Java プログラミング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．

- jess

  jess とは Java Expert Shell System の略である．このプログラムは，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．

- db

  db はメモリ上に存在するデータベースに対する操作を行うものである．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．

- javac

  javac は Java Development Kit(JDK) 1.0.2 の Java コンパイラである．

- mpegaudio

  mpegaudio は MPEG 3 オーディオをデコードするものである．

- mtrt

  mtrt はレイトレーシングを行うものである．このプログラムにおいては，2 つのスレッドがそれぞれレンダリングを行う．

- jack

  jack は Java プログラミング言語のパーザジェネレータである．

また，測定には Kaffe 1.0b4 を用いた．ソフトウェアにより実装されたこの Java バーチャルマシンに VLB を操作するためのコードを追加する形で，シミュレーションを行った．ハードウェアにおけるデータ再利用の評価も，実際のハードウェアではなくこの Java バーチャルマシンを用いて行った．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．

5.2 VLB の効果の限界

Java バイトコードを直接実行した場合におけるデータ再利用の効果の最大値を調査した．本節では，データ再利用に要する処理時間を 0 と仮定した場合に，データ再利用により高速化が可能である実行時間を示している．

この評価には，VLB を用いないで実行した場合の全実行命令数に対する，VLB を用いることにより実行を省略することが可能であった命令数の比を用いている．このため，インタプリタ方式かハードウェア直接実行かの違いによる差はない．また，VLB の総エントリ数は 32768 とした．1 つの VLB のエントリに登録することが可能なヒープアクセスの回数は十分にとり，VLB に登録することが可能な命令群の種類をなるべく多くした．

5.2.1 区切り命令を用いる場合

区切り命令を用いる場合に実行が省略された命令数の比を表 2に示す．区切りとなる命令間の命令数が 4 以下の場合は VLB への登録を行わないこととした．この場合の効果は 0.1% から 14.2% であり，mtrt の結果が極端に低い．

5.2.2 メソッドを単位とする場合

メソッドを単位とする場合に実行が省略された

表 2: 区切り命令を用いる場合に実行が省略された命令数の比 (単位は%)

	compress	jess	db	javac	mpegaudio	mtrt
s1	5.32	11.0	14.2	12.1	3.51	1.96
s10	3.51	1.21	5.96	9.37	2.38	0.514
s100	5.27	4.54	4.66	9.59	3.16	0.0552

表 3: メソッドを単位とする場合に実行を省略された命令数およびメソッド数の比 (単位は%，上段は命令数の比，下段はメソッド数の比)

(%)	compress	jess	db	javac	mpegaudio	mtrt
s1	3.86 (23.5)	13.0 (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)

命令数およびメソッド数の比を表3に示す．表の上段は，命令数の比である．表の下段の括弧内の数値は，データ再利用を行わない場合の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を省略されたメソッド数の比である．その効果は，命令数の比においては0.1%から26.0%である．メソッド数の比においては0.1%から58.3%である．この値は，区切りとなる命令間の命令群を単位とする場合の結果と比較して，大きくばらついている．

5.3 ハードウェアにおける効果

4の方針に従って，VLBをハードウェア化した場合の効果を測定した．

区切り命令を用いる場合，実行結果の再利用が可能であったとしても，4で述べたように少なくとも3サイクルのパイプラインハザードが生じる．5.2.1項における結果は，そのパイプラインハザードが無視できるほどこの方針によるデータ再利用の効果が高いとは言えないことを示している．

一方，メソッドを単位とする場合においては，メソッド呼び出しおよびリターンに伴うオーバヘッドを削減することが可能である．このため，VLBをハードウェア化する場合，メソッドを単位とする手法は効果的である．従って，以下ではメソッドを単位としたVLBのハードウェア化を前提として評価をすすめる．

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サイクルにより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つのヒープ上の値と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，0.1%から47.1%のサイクル数を削減することが可能である．

5.4 VLBのエントリ数

データ再利用の性質上，VLBに登録できるエントリ数には限度がある．そこで，エントリ数とデータ再利用の効果の関係を調べることにより，最適なVLBのエントリ数を決定する．

VLBに登録可能な総エントリ数を4096とした場合に，データ再利用により削減可能となった実行サイクル数を表5に示す．この結果と表4の結果

表 4: ハードウェア化した VLB を用いた場合の省略サイクル数 (単位は%)

(%)	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 5: VLB のエントリ数を 4096 とした場合の削減可能なサイクル数 (単位は%)

(%)	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

を比較すると，compress 以外のベンチマークにおいては比較的近い数値が得られた．このことより，データ再利用により削減可能となった実行サイクル数が，これらのベンチマークプログラムがどの程度データ再利用に適しているかを表していると考えられる．これらのベンチマークプログラムは，登録可能なエントリ数を増加しても，削減可能なサイクル数が大幅に増加するとは考えられない．

しかし，エントリ数の影響を受けた compress は，VLB に登録可能なエントリ数を増加させることによりより大きな効果が現われる可能性がある．そこで，compress を実行した時の VLB のエントリ数と削減可能なサイクル数の関係について調査を行った．その結果を図 16に示す．図の横軸は VLB の総エントリ数，縦軸はその時に削減可能となった実行サイクル数である．

この結果によると，登録可能なエントリ数を増加しても，データ再利用の効果はある一定の値に収束すると考えられる．またこの結果より，VLB の総エントリ数は 32768 エントリもあれば十分な効果が得られるものと考えられる．

5.5 考察

SPEC JVM 98 のベンチマークプログラムのうち，mtrt は浮動小数点数を多く扱う．使用される浮動小数点数の種類は，整数と比較して膨大となりやすい．この結果，登録すべき実行結果と比較して，実際に VLB に登録される実行結果は非常に小さなものとなってしまう．これは s1 の mtrt をメソッドを単位として行った場合以外の全ての評価結果に当てはまる．すなわり，データ再利用と浮動小数点数演算の相性は良くないと言える．一方，メソッドの実行を単位とする場合における s1 の mtrt の結果は非常に高い値が得られている．これは，このプログラムが表 1に示すようにメソッドの呼び出しを多く含み，データ再利用に適した記述が行われているからであると考えられる．

mpegaudio は比較的省略可能なメソッド数が多い．しかし，省略可能な命令数は比較的少ない．これは，mpegaudio において省略可能なメソッドにより実行されるバイトコードが短いからである．しかし，表 1に示された命令の出現頻度によると，メソッド呼び出し命令は他のベンチマークプログラムと比較して実行される割合が低い．つまり，平均するとメソッドは長いと言える．このことより，mpegaudio のメソッドはバイトコードの長いものと短いものの差が極端であると言える．このうち，短いものがデータ再利用可能となっている．

javac は区切り命令を単位とする場合，他のベンチマークプログラムと比較してデータ再利用の効果は高い．しかし，表 1に示された命令の出現頻度によるとメソッドの呼び出しが多数発生するにも関わらず，メソッドを単位とする場合，その効果は他のベンチマークと比較して高いとは言えない．javac は，他のベンチマークプログラムと比較して，多重にメソッドを呼び出す．バイトコードの実行中にメソッド呼び出しが発生すると，メソッド呼び出しからリターンまでに実行するバイトコードは増加する．つまり，データ再利用の単位が長くなってしまう．javac がメソッドの呼び出

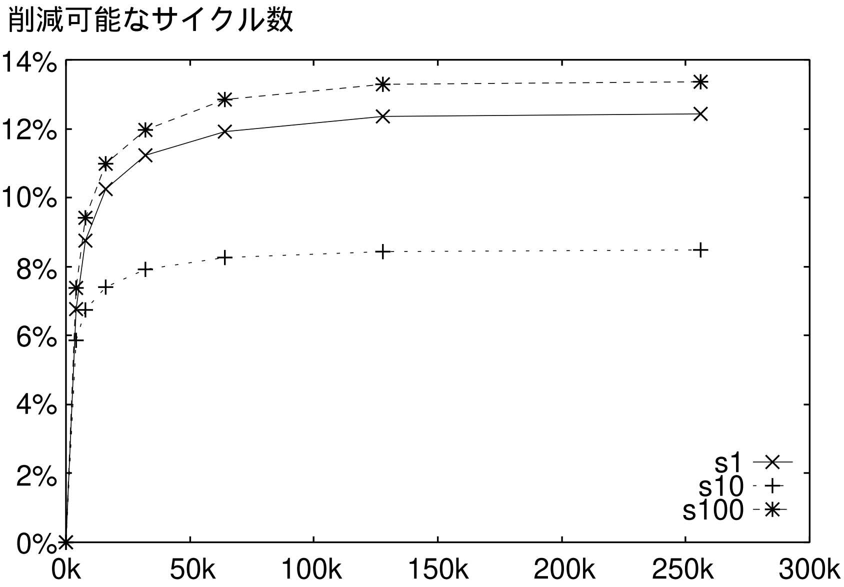


図 16: VLB のエントリ数と削減可能なサイクル数の関係

しが多数発生するにも関わらず，メソッドを単位とする場合のデータ再利用の効果が高くないのは，このためである．

Java コンパイラは，インライン展開を行うことができない．これは，実行するメソッドが動的に決定されるからである．しかし，Java バーチャルマシンにおいて実行されるメソッドの中には，何も行わない，あるいは定数を戻り値として渡す，などの簡単な処理を行うものが比較的多い．このような場合，区切り命令を用いたデータ再利用では，命令群の命令数が少ないため効果はない．しかし，メソッドを単位とする場合には，メソッド呼び出しのオーバヘッドを削減することが可能である．このため，このように単純なバイトコードから成るメソッドであればあるほど，比較の結果が一致する場合が多く，データ再利用の効果は高い．

区切り命令を用いる場合は，浮動小数点数演算を行う mtrt を除いて，ベンチマーク毎のデータ再利用の可能な命令数の比率がほぼ同じある．命令数で比較した場合，区切り命令を用いる場合において最大の効果を示す s1 の db でさえも，メソッドを単位とするものにおいて最大の効果を示す s100 の jess における効果のおよそ半分程度である．またメソッドを単位とする場合は，メソッド呼び出しのオーバヘッドの軽減も可能である．このことより，区切り命令を用い細かい単位のデータ再利用を行うよりも，メソッドを単位としてデータ再利用を行う方が効果が高いと言える．

メソッドを単位とする場合には，効果に大きなばらつきがある．効果が最も良く現れたのは jess である．これは，jess のソースプログラムの記述がオブジェクト指向を強く意識しているためと考えている．

オブジェクト指向型プログラミング言語においては，データとデータを扱うメソッドの組み合わせであるオブジェクトが動作の単位である．ところが，Java プログラミング言語においては，int などの基本的な型はクラスとして定義されていない．例えば int 型の変数 1 つであっても，これに対する操作をメソッドとして記述することによりオブジェクトとすることが可能である．一方，これをオブジェクトとせず，メソッド内のローカル変数として記述することも可能である．ローカル変数とした記述を繰り返すことにより，メソッドの記述が繁雑となり，処理量が増加する．この結果，ヒープ上の値の参照回数が増加するために，デー

タ再利用が効率良く行われない．

またオブジェクト指向型プログラミング言語においては，あるオブジェクトが他のオブジェクトの内部情報を知る必要はない．メソッドの記述において他のオブジェクトの内部情報を直接参照するという方法を用いると，メソッドの実行中に生じるヒープアクセスの回数が多くなる．このため，入力データの種類が増加してしまう．この結果，データ再利用の効率が低下する．またこのような記述はソースコードの再利用性が低いため望ましいものではない．他のオブジェクトの内部情報を得るためには，オブジェクトの属するクラスに他のオブジェクトに内部情報を通知するためのメソッドを用意し，それを使用して行うべきである．このように記述することにより，メソッドの実行時に参照するヒープ領域が少なくなり，データ再利用を効果的に行うことが可能となる．

一般的にオブジェクト指向を意識したプログラムは，そうでないプログラムよりもメソッド呼び出しの回数が多く，実行速度は低下する．しかし，データ再利用の手法を用いることにより，オブジェクト指向を意識しないプログラムと同等あるいはそれ以上に高速に動作することが可能である．

5.6 今後の方針

命令畳み込みとの組み合わせ

4.1節で述べたプロセッサは，命令畳み込み[12]を考慮に入れ設計されたものである．この命令畳み込みとVLBを使用したデータ再利用との関係はどのようになるのであろうか．メソッドを単位とした場合についてのみ考える．

メソッドを単位とするデータ再利用の場合，実行するバイトコード数の減少および呼び出すメソッド数の減少により高速化が達成される．命令畳み込みは，プロセッサ内部で発行する命令数を減少することにより高速化を行う手法である．この効果は，データ再利用における実行命令数の減少の効果と重複してしまい，命令畳み込みの方がデータ再利用よりも大きいことがわかっている．

一方データ再利用は，メソッド呼び出しを省略することにより高速化することが可能である．つまり命令畳み込みとデータ再利用を組み合わせた場合，命令畳み込みによる実行命令数の減少による高速化に加えて，データ再利用によるメソッド呼び出し数の減少による高速化が期待できる．このため，命令畳み込みとデータ再利用の組み合わせは効果的であると言える．

データ再利用可能判定処理の高速化

本稿においては，参照するメモリの値を直接比較することによりデータ再利用が可能であるかどうかを判断している．しかし値を比較するためには，メモリアクセスが必要となる．このメモリアクセスのうちヒープアクセスは，アドレス変換などの処理を必要とする．このため，本稿において述べた手法は，比較を行うために長い時間を必要とする．

そこでこの比較の時間を短くする方法として，実際にメモリの値の比較を行わない方法が考案されている[8]．これは，前回メモリアクセスを行った時点からその値が変更されていないことが保証されていれば，実行結果を再利用するという手法である．

この手法は図17に示すように，メモリのリードが発生してもその値の登録は行わない．その代わりに，このアドレスを参照して行われた実行結果へのポインタを保持しておく．このアドレスに対して更新が行われた場合，その値を参照して求められた実行結果を全て無効化する．このような処理を繰り返すことにより，データ再利用の可能な実行結果のみが有効なエントリとして登録される．これら有効なエントリは実行結果を再利用する際に比較を行う必要がないため，高速に判断することが可能である．以下に，この手法をJavaバーチャルマシンに実装した場合について述べる．便宜上，実行結果を保存するメモリをVLBと呼ぶことにする．

ヒープ領域のみをこの手法の対象とする．これは，オペランド・スタックおよびローカル変数領域はメソッドの呼び出し毎に新しく生成されるためである．これらの領域は，実際の値を用いて比較を行う．

VLBにはヒープのリードについての情報を登録しない．代わりに，ヒープ上のアドレス毎にVLBのインデックスのリストを保持する．ヒープのリードが発生すると，このリストに現在登録中のVLBへのインデックスを登録する．ヒープへのライト

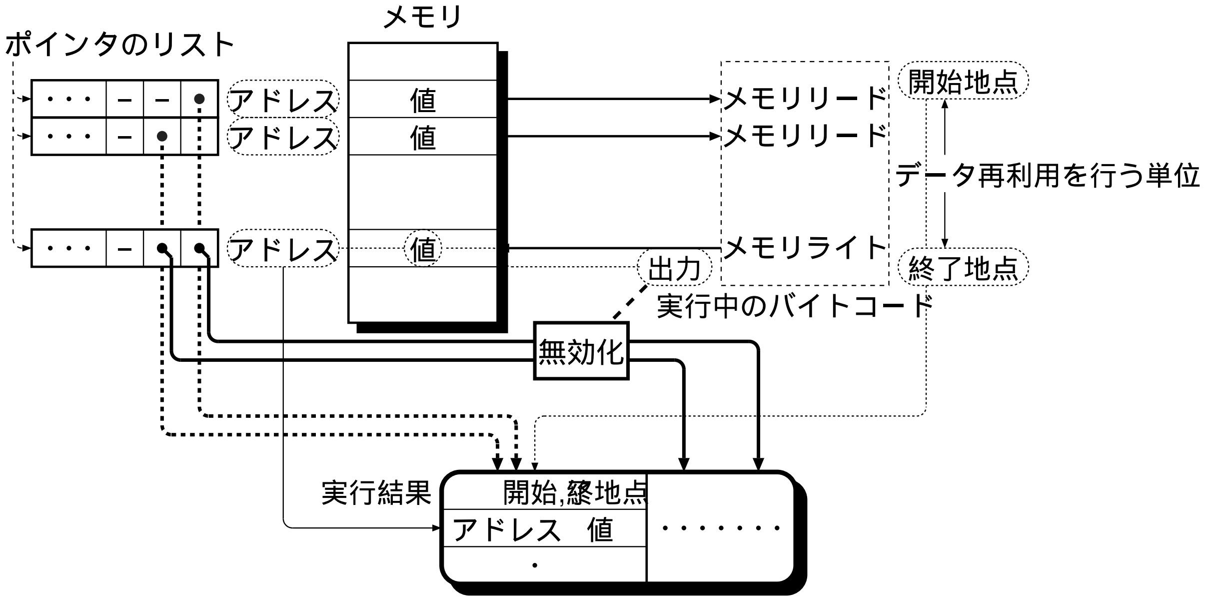


図 17: 値を直接比較することなくデータ再利用可能判定を行う手法

が発生すると，そのアドレスに対応するリストに登録されたVLBのエントリを全て無効化する．また，削除を行ったリスト以外のリストから，無効化の対象となったVLBのエントリへのインデックスを削除する．これは，無効化の対象となったVLBのエントリを再利用するためである．このような手順をふまえることにより，データ再利用が可能であるかどうかの判断を，オペランド・スタックおよびローカル変数上の値の比較のみで行うことが可能となる．

この手法は比較に必要となる処理時間を減少する代わりに，VLBのエントリの無効化に要する処理時間を増大させる．またこの手法においては，無効化が完了するまでどのVLBのエントリが有効であるかを知ることはできない．つまり，無効化が完了するまで，データ再利用を行うことができない．従って，ヒープの値の更新が行われてからデータ再利用が可能であるかどうかの判断までの時間が短い場合には，実行結果を再利用することが不可能である．このような状況が多数発生すれば，データ再利用による高速化が十分に行われない．また，1つの命令群につき1つの実行結果のみ登録することが可能であるため，VLBに登録可能となる実行結果の種類は非常に限定されてしまう．このことより，直接値を比較しない手法を用いても，データ再利用の効果が増大するとは考えにくい．また，無効化のための複雑なメモリ機構が必要となる．極めて高い効果が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．

一方，値を直接比較するデータ再利用の手法をハードウェア化した場合，再利用が可能であるかどうかの判断に必要となる処理時間は十分存在した．また測定を行った結果，本稿で述べた手法を用いた場合にもデータ再利用の効果は十分に得られることが判明した．以上のことから，値を直接比較することによりデータ再利用を行う手法が，無効化の行われなかった有効なエントリを用いてデータ再利用を行う手法よりも効果的であると言える．

6 おわりに

本稿では，データ再利用を用いてJavaバイトコードの実行速度を高速化する手法について詳述した．

バイトコードの実行時にデータ再利用の手法を行う単位として，区切り命令を用いる方法，およびメソッド単位とする方法の2つを提案した．そ

れぞれの手法を，インタプリタ方式による Java バーチャルマシンに実装した結果，大きなオーバヘッドのために，インタプリタ方式として高速化することはできなかったものの，Java バイトコードには再利用による高速化の可能性があることが判明した．

そこで，VLB をハードウェア化しデータ再利用を実行することにより，VLB の操作に必要となる処理時間をバイトコードの実行時間に隠蔽した場合についての調査を行った．区切り命令を用いる場合には，データ再利用が可能であってもパイプラインハザードが発生するため効果が得られない．そこで，ハードウェア上でのデータ再利用は，メソッドを単位とすることとした．

VLB の効果をベンチマークプログラムを使用して調査を行った結果，区切り命令を用いる場合よりも，メソッドを単位とする場合の方が良い結果を示した．しかし，メソッドの実行を単位とする場合の効果には大きなばらつきが存在する．このばらつきの原因は，実行したベンチマークプログラムの記述方法によるものである．オブジェクト指向を強く意識して書かれたプログラムに記述されたメソッドは，データ再利用の対象となる場合が多いため，高速に実行することが可能である．

VLB をハードウェア化する場合，メソッドの呼び出しおよびリターンという処理は，パイプラインハザードを伴う．メソッドを単位としたデータ再利用の手法をハードウェア化することにより，パイプラインハザードに伴うサイクル数を減少させることが可能となり高速化が可能である．

今後は，他の高速化手法との組み合わせについて考える予定である．また，投機的実行のとの組み合わせの可能性についてもさらに検討する予定である．

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．

数々の有用な御指導，御意見を頂いた森 眞一郎助教授，中島 康彦助教授，五島 正裕助手に心から感謝致します．

本研究の共同研究者である，山田 克樹氏，小川洋平氏をはじめ，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．

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

参考文献

[1] 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, 情処研報 (2000)（発表予定）

[2] J. ゴスリン, B. ジョイ, G. スティール (村上訳):The Java 言語仕様, アジソン・ウェスレイ・パブリッシャーズ・ジャパン社 (1997).

[3] T. リンドホルム, F. イェリン (野崎訳):The Java 仮想マシン仕様, 言語仕様, アジソン・ウェスレイ・パブリッシャーズ・ジャパン社 (1997).

[4] J. メイヤー, T. ダウニング (鷲見訳): JAVA バーチャルマシン, オライリー・ジャパン社 (1997).

[5] A. Sodani, G. S. Sohi: Understanding the Differences Between Value Prediction and Instruction Reuse, 31th International Symposium on Microarchitecture(1998).

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

[7] S. E. Richardson: Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation, SMLI TR-92-1, Sun Microsystems Laboratories (1992).

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

[9] A. González, J. Tubella and C. Molina: Trace-Level Reuse, 1999 International Conference on Parallel Processing (Hardcover) (1999).

[10] SPEC JVM 98 VERSION 1.03, the Standard Performance Evaluation Corporation (1998).

[11] 木田 裕之, 木村 晋二, 高木 一義, あべ松 竜盛, 渡辺勝正: 再構成可能部を持つ Java プロセッサ, 電子情報通信学会総合大会, pp. 442-443 (1999).

[12] L. -R.Ton, L. -C.Chang, M. -F.Kao, H. -M.Tseng, S. -S. Shang, R. -L. Ma, D.-C.Wang and C. -P.Chung: Instruction Folding in Java Processor, 1997 International Conference on Parallel and Distributed Systems (1997).