情報処理学会論文誌　コンピューティングシステム　Vol. 4　No. 3　115–132 (May 2011)

Android 端末におけるハードウェアによる Java の高速化手法の提案

太田　淳[†1]　三輪　忍[†1]　中條拓伯[†1]

Android 端末では，Java プログラムは，Dalvik バイトコードと呼ばれる独自のバイトコードに変換され，VM を介して実行される．VM による実行は時間がかかるため，Java バイトコードを携帯端末で実行する場合は，ハードウェア・アクセラレーションがよく行われる．一方，Dalvik バイトコードの場合は，まだ歴史が浅いため，その高速化に関する研究は十分でない．そこで我々は，携帯端末における Dalvik バイトコード実行の高速化機構として，Dalvik アクセラレータを開発することにした．バイトコードの各オペランドはメモリ上に存在するため，単純にアクセラレータを実装すると，多数のメモリ・アクセスが発生してしまう．この問題に対し，物理レジスタを最大限活用することでメモリ・アクセスを削減する機構を提案する．本機構により，大部分のメモリ・アクセス命令を削減できることが分かった．

Proposal of a Hardware Scheme for Java Acceleration on Android Devices

Atsushi Ohta,[†1] Shinobu Miwa[†1] and Hironori Nakajo[†1]

On Android devices, a Java application is compiled to a specific bytecode called as Dalvik bytecode, then, the bytecode is executed on VM. Since the execution on VM has large overhead, in case of Java bytecode, a hardware acceleration is often used on mobile devices. On the other hand, acceleration techniques for Dalvik bytecode have not been studied extensively because the bytecode is launched recently. Therefore, we develop a Dalvik accelerator: a hardware mechanism for Java acceleration on Android devices. Simple implementation of the accelerator causes large amount of memory accesses, because every operands of a Dalvik bytecode exist in a main memory. Therefore, we propose a mechanism of reducing memory accesses using physical registers. This paper shows that the mechanism reduces large amount of memory access instructions.

1. はじめに

近年，Google 社の携帯情報端末向けプラットフォーム，Android[4),5)] を搭載した機器が急速に普及している．携帯電話では，Xperia，Droid，NexusOne に代表されるように，Android 携帯が多くのメーカから次々と市場に投入され，一定のシェアを占めるに至っている．また，セットトップボックスやネットブックにも搭載されるなど，Android の用途は急速に広まっている[22)]．

Android 上のアプリケーションは，ポータビリティを高めるため，普通は Java 言語で記述され，バイトコードにコンパイルされる．バイトコードはホスト・プロセッサに非依存であるため，それを解釈/実行できる環境があれば，任意のプロセッサ上で実行できる．通常，VM によって逐次解釈されつつ実行される．

ただし，Android における Java の実行モデルは，通常の Java のそれとは若干異なる．Android ではバイトコードとして，通常の Java バイトコードではなく，Dalvik バイトコード[2)] と呼ばれる独自の命令セットを採用する．両者は，前者がスタック・マシンを想定した命令セットなのに対し，後者はレジスタ・マシンを想定したものとまったく異なる．そのため，バイトコードの実行にも，通常の Java VM ではなく，独自の VM である Dalvik VM[2)] が用いられる．

バイトコードの実行は，上述のように VM を介して行われるため，ネイティブ・コードの実行と比べて遅い．そのため，Java バイトコードに対しては，その実行を高速化する手法がこれまでに提案されてきた．

Java バイトコードに対する高速化手法

ソフトウェアによる高速化手法としては，実行時コンパイル[14),19)]（Just-In-Time コンパイル．以下 JIT とする）や事前コンパイル[16)]（Ahead-Of-Time コンパイル．以下 AOT とする）がある．これらの方法は，バイトコードの一部もしくは全部を，実行時もしくは実行前にネイティブ・コードへコンパイルする．コンパイルされた部分に関しては，プロセッサは高速に実行できる．実際，汎用 PC で動作する Java VM で JIT や AOT が用いられている．ただしこれらの手法には，詳しくは 5.4 節で述べるが，コードがコンパイル前に比べ

†1 東京農工大学
Tokyo University of Agricalture and Technology

て膨らんでしまうという欠点がある．これは大きなメモリを搭載できない組み込みプロセッサにおいて，おおいに問題である．

そこで，ハードウェアによるアクセラレーションが解決策として利用される．プロセッサの一部やコプロセッサとして，バイトコードを解釈/実行できるハードウェアを追加する．VM を介さないことで，オーバヘッドを減らすことができる．バイトコードを直接実行することから，JIT や AOT のようにメモリを圧迫しない．アクセラレータの回路規模を小さくできれば，組み込みシステムにおいては有効な選択肢となる．

実際，Java アクセラレータ[10),13)] は多くの携帯電話に搭載されている．なかでも，ARM 社の Jazelle DBX[1),15),18)] は，追加ハードウェア量が少ないアクセラレータとして知られる[9)]．**Jazelle 方式**では，プロセッサの既存のデコーダに加え，バイトコードを解釈するデコーダを追加する．アクセラレーションを行う際は，この専用デコーダがバイトコードをネイティブ・コードへ変換し，後続のユニットへ命令を発行する．演算器やレジスタなど，既存のプロセッサ資源をほぼそのまま流用するため，追加するのはデコーダだけでよい．

Dalvik バイトコードに対する高速化手法

一方，Dalvik バイトコードは登場してまだ間もないこともあり，その高速化手法についてはほとんど研究されていない．最新の Dalvik VM になってようやく，試験的に JIT が組み込まれた程度である．上述のように，携帯電話のアプリケーション・プロセッサではハードウェア・アクセラレーションが有効な手段と考えられるが，我々の知る限り，Dalvik VM のアクセラレータを実現した例はない．レジスタ・ベースの VM はスタック・ベースの VM よりも高速だと考えられている[17)]が，上述のような背景から，Dalvik VM 上でのアプリケーション実行は Java VM の場合と比べて 10 倍も遅くなってしまうこともある[25)]．

そこで我々は，Dalvik VM のハードウェア･アクセラレーションを試みることにした[23),24)]．アクセラレーション方式は，Java VM において有力な Jazelle 方式を採用する．このようにすることで，追加ハードウェア量が少ない Dalvik アクセラレータの実現を目指す．

Dalvik バイトコードのオペランドはメモリ上に存在するため，バイトコード 1 命令を単純に変換すると，そのつど，各オペランドのロード/ストアを行ってしまう．この問題に対し，我々は **DRMT（Dalvik Register Map Table）**を提案する．DRMT は，物理レジスタにマップされたオペランドを管理する表である．これを利用して，物理レジスタにマップされたオペランドの，ロード/ストアの発行を抑止する．

本稿の構成は以下のようになっている．まず次章において，Java VM について簡単に述べた後，Jazelle 方式について詳しく説明する．続く 3 章では，Java VM と比較しながら Dalvik VM アーキテクチャを概観する．4 章で提案する Dalvik アクセラレータと DRMT について説明し，5 章で評価を行い，6 章でまとめる．

2. Java VM と Jazelle 方式

Java VM は，Java バイトコードの実行をはじめ，GC やベリフィケーションなど，さまざまな処理を行っている．Java アクセラレータは，これらの中で最も負荷が高い，バイトコード実行をハードウェアで行うことで，高速化を図る[⋆1]．以下，まずバイトコード・インタプリタを概観し，次いで Jazelle 方式[1),3),7),9),15),18)] について詳しく述べる．

2.1 Java バイトコード・インタプリタ

Java バイトコードは，スタックの push/pop により演算を行う，スタック・マシンを想定した命令セットである．演算対象はスタックの上部に限定されるため，演算命令はオペランドを持たないのが特徴である．そのため，命令の種類によって長さが異なり，5 バイト以内の可変長命令セットである．

インタプリタが Java バイトコードを実行する様子を**図 1** に示す[6),28)]．インタプリタは，以下の 3 種類の記憶領域[⋆2]を用いてバイトコードを処理する．

(1)　**メソッド領域**
(2)　**ヒープ**
(3)　**Runtime Data Area**

以下，それぞれについて詳しく述べる．

2.1.1 メソッド領域

メソッド領域は各クラスの実装に関する情報を保持する領域である．クラスが初めて参照されるたびに連続する領域が割り当てられ，対応するクラス・ファイルの内容がローダによってそこに読み込まれる．図は A，B の 2 つのクラス・ファイルが展開された状態である．

クラス・ファイルは以下の情報を含む．

コンスタント・プール　クラスで使用される静的な情報を集めた領域．定数値だけでなく，他のクラスの参照などのオブジェクトの参照も含まれる．

インタフェースに関する情報　インタフェースを実装したクラスである場合は，インタフェースの数，実装などに関する情報が記録される．

⋆1 GC もハードウェアで行うアクセラレータも一部ある[8)]．
⋆2 図ではメソッド領域とヒープを分離しているが，ヒープ上にメソッド領域を確保する実装もある．

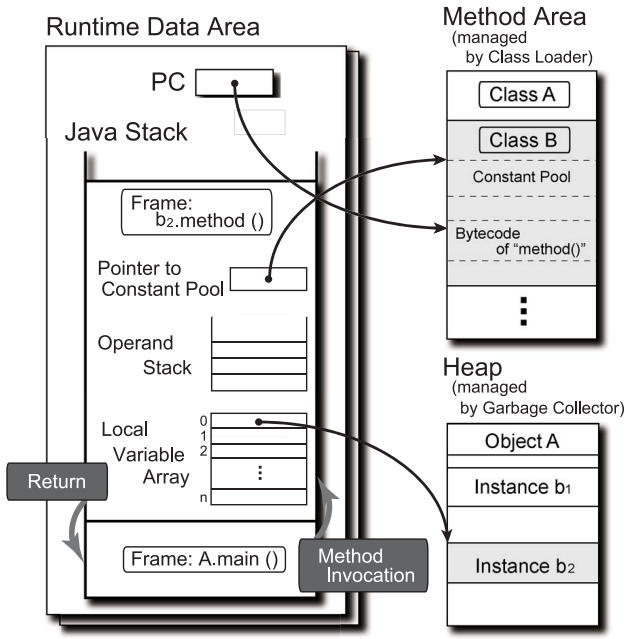

図 1　インタプリタによる Java バイトコード実行
Fig. 1　Java bytecode execution by interpreter.

メンバ変数に関する情報　クラスで定義されているメンバ変数の数や型などの情報．そのメンバ変数の値が，実際にここに格納されるわけではないことに注意されたい（値の格納場所はヒープ領域）．

メソッドに関する情報　クラスで定義されているメソッドの数や型の情報．各メソッドの情報には，後述するように，メソッド内で使用されるローカル変数の数やオペランド・スタックの最大値などが含まれる．また，各メソッドの命令列もここに記述されている．したがって，インタプリタは，ここに展開された命令列を見て，1 命令ずつ処理を進めることになる．

2.1.2　ヒ　ー　プ

ヒープはオブジェクトの値を保持する領域である．生成されたインスタンスの値を保持するための領域がヒープに作られる．図 1 では，クラス B のインスタンス b_1，b_2 それぞれに対し，異なる領域が割り当てられている．割り当てられた領域の回収は GC が行う．

インタプリタは，インスタンス・メンバ変数の値をここから読み出し，更新した値をここに格納する．

2.1.3　Runtime Data Area

Runtime Data Area は，ローカル変数などの一時的な情報を保持する領域である．上述の 2 つの領域は VM 全体で 1 つしか存在しないのに対し，Runtime Data Area はスレッドごとに独立して存在する．

Runtime Data Area は PC とコール・スタック（Java スタック）からなる．PC はインタプリタが現在処理中の命令を指すポインタであり，メソッド領域上の該当する命令のアドレスが格納される．コール・スタックは後述する**フレーム**を管理しており，メソッドを呼び出すたびにフレームを push し，復帰するたびに pop する．図は，main 関数からインスタンス b_2 の method 関数を呼び出した状態を表す．フレームは以下の要素からなる．

コンスタント・プールへのポインタ　前述のように，メソッドで使用される定数――定数値やオブジェクトの参照――はコンスタント・プールに存在する．そのため，そこへのポインタを必要とする．

オペランド・スタック　いわゆる Java VM のスタック．オペランドをここに push/pop して演算する．

ローカル変数配列　メソッドで使用されるローカル変数を保持する配列．メソッド引数もここに保持される．インスタンス・メソッドでは，ローカル変数の 0 番はつねにオブジェクトの参照に対応する．

2.1.4　全体の処理の流れ

VM が起動されると，まず，エントリ・ポイントを含むクラス（図 1 では “Class A”）のクラス・ファイルがメソッド領域に展開され，初期化が行われる．同時に，ヒープ上に領域（“Object A”）が確保される．Runtime Data Area を生成し，main メソッドのフレームをスタックに積む．そして，PC に main メソッドの先頭の命令のアドレスをセットし，実行を開始する．

インタプリタは，オペランド・スタックを用いて，PC が指す命令を実行する．命令に応じて必要なオペランドをスタックに積み，オペコードに従って演算を施す．オペランドは，ローカル変数配列，ヒープ，コンスタント・プールから取得する．

新たなメソッドを呼び出す命令（`invoke_*` 命令）を実行すると，インタプリタは，フレームを生成してスタックにそれを積む．また，新たなオブジェクトを生成する命令（`new` 命

令）を実行すると，まず GC プログラムを呼び出し，そのオブジェクトの領域を割り当てる．クラスがロードされていない場合は，ローダにより対応するクラス・ファイルのロードが行われる．

このように，インタプリタは，命令によっては他のソフトウェア・モジュールとの協調を必要とする．

2.2　Jazelle 方式

Jazelle 方式を適用した場合のプロセッサ構成を**図 2** に示す[⋆1]．Jazelle 方式では，バイトコード実行のためのデコーダをプロセッサに追加する．バイトコード 1 命令がフェッチされると，デコーダが同じ意味を持つ命令列へと変換する．変換された命令列はすべてネイティブ・コードであるため，新たなレジスタ/実行ユニットを追加することなく，既存の資源を利用して実行できる．

Jazelle 方式では解釈する命令セットに対応して 2 つの実行モードが存在する．ネイティブ・コードのデコーダが有効な ARM モード，Java バイトコードのデコーダが有効な Jazelle モードである．実行モードをステータス・レジスタにより識別し，2 つのデコーダが選択され，プロセッサが解釈する命令を切り替える．

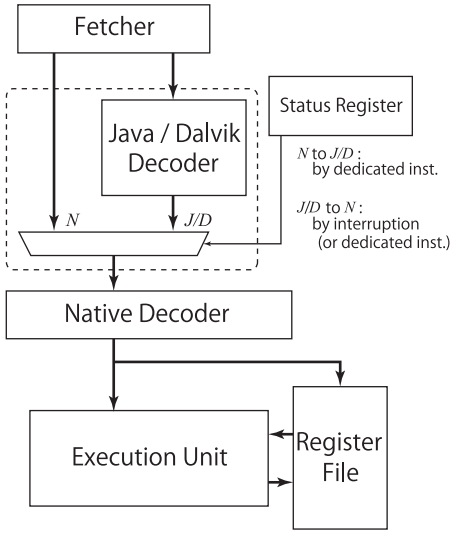

図 2　Jazelle 方式のブロック図
Fig. 2　Block diagram of Jazelle method.

⋆1 Jazelle DBX の詳細は公表されていないため推測ではあるが，ARM 社の実装では，2 つのデコーダを並列に配置しているようである[15),18)]．図は，Capewell らの実装[3)] を参考にした．

ステータス・レジスタを変更し，Jazelle モードへ切り替えるには **BXJ（Branch and change to Jazelle state**）命令を使用する．BXJ 命令は，無条件分岐とともにステータス・レジスタを Jazelle モードに変更する命令である．分岐先アドレスには実行する Java バイトコードのアドレスを指定する．

Jazelle モードに切り替わると，デコーダは Java バイトコードのみを解釈する．Java 例外の発生か，デコーダでは変換できないバイトコードが入力されると，ARM モードへ戻る．

Jazelle DBX における ARM レジスタの使用方法を**表 1** に示す．PC（r15），あるいは，コンスタント・プールへの参照（r8）やローカル変数への参照（r7）など，高頻度で使用される値がレジスタに静的に割り当てられる．これらの値は，インタプリタ実行の際も，当該レジスタに割り当てられる．したがって，ARM モードから Jazelle モード，あるいは，その逆方向に遷移する際，これらのレジスタの値を退避する必要はない．高速にコンテキスト・スイッチできる．

また，オペランド・スタックの先頭データ数個分もキャッシュする（r0〜r3）．スタックの上位にあるオペランドについては，レジスタに対して直接操作できる．したがって，オペランドを操作する際のロード/ストアを削減できる．

前述のように，Java バイトコードの 1 命令は，命令によっては ARM 1 命令よりも長いため，フェッチに複数サイクルを要する場合がある．この場合，フェッチすべきアドレスと PC（処理中のバイトコードの先頭アドレス）は一致しない．この問題を解消するため，表には明記していないが，ARM レジスタ r15 に対応する PC とは別に，フェッチ用の PC が存在する．

表 1　Jazelle DBX におけるレジスタ割当て（一部）
Table 1　Portion of register map in Java mode.

番号	用途
r0〜r3	オペランド・スタックの先頭 4 つの値
r4	this ポインタ
r5	ARM モードへ復帰時のハンドラへのポインタ
r6	オペランド・スタックへのポインタ
r7	ローカル変数へのポインタ
r8	コンスタント・プールへのポインタ
r15	Java プログラム・カウンタ

バイトコードの中には，ハードウェアだけで処理するのが難しい命令もある．そうした命令については，ARM モードへ制御を戻し，通常どおり VM で処理する．Jazelle DBX では，以下の命令は，デコーダによる変換の対象としない[9]．

ベース・プロセッサの構成上実行できないもの

- 整数除算
- 浮動小数点演算

他のモジュールとの協調を必要とするもの

- コンスタント・プール上のデータのロード
- オブジェクトに対する操作
- メソッドの invoke と return

その他複雑な処理を必要とするもの

- テーブル・ジャンプ
- ロックの獲得と解放
- 例外処理など

このように Jazelle 方式では，ハードウェアによる処理とソフトウェアによる処理を切り替えながら，バイトコード実行を進める．ARM 社によると，この方式により，バイトコードの全実行時間のうち，95%以上がハードウェア実行できる[18] としている．

3. Dalvik アーキテクチャ

Dalvik VM はレジスタ・マシンの VM[26] であることから，Java VM と内部仕様が異なる部分が存在する．本章では，Dalvik VM における相違点について示す．

3.1 バイトコードの仕様

Java バイトコードと同様，Dalvik バイトコードもまた，命令は可変長である．2 バイト単位で，2〜10 バイトの長さを持つ．先頭から数えて 1 バイト目のフィールドがオペコードに相当する．各命令は最大 5 個のオペランドを持つ．オペランドのうち，変数のものは **Dalvik レジスタ**と呼ばれる．例として，**図 3** に 3 つの Dalvik レジスタ（vA，vB，vC）をオペランドとして持つ `add-int` 命令と，6 バイト幅で，フェッチ幅が 4 バイトのプロセッサでは 1 サイクルでフェッチできない命令例として `move-wide` 命令を示す．1 つのブロックは 1 バイトであり，`move-wide` 命令は計 6 ブロックに相当する．

3.2 Dalvik レジスタ

Dalvik バイトコードでは，各命令は Dalvik レジスタに対して演算を施す．Dalvik レジスタは 4 バイト幅で，メソッドごとに仕様上は 65,536 個のレジスタを使用できる．ただし，Dalvik レジスタはローカル変数に相当するため，実際のプログラムでは大量の Dalvik レジスタが使われることはほとんどない．実際，Android のソースコードに含まれる標準アプリケーションでは，使用する Dalvik レジスタが 12 個以内のメソッドが 94%を占める．各命令は，0〜5 個のレジスタをオペランドとし，それらの値を用いて演算する．

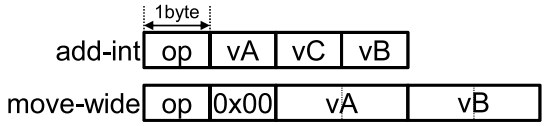

図 3　Dalvik バイトコードの例
Fig. 3　Example of Dalvik bytecode.

ここで，Dalvik レジスタは，ハードウェア・レジスタとは別物である点に注意されたい．レジスタと呼ばれているが，実際には単にメイン・メモリ上に配置された配列データにすぎない．そのため，インタプリタが Dalvik レジスタを操作する際は，演算する前にまずは該当するデータをロードし，演算結果をストアする，という処理が行われる．

3.3 Interpreter State

Dalvik インタプリタが Dalvik バイトコードを実行する様子を**図 4** に示す．Dalvik VM も Java VM 同様に，以下の 3 種類の記憶領域からなる．

(1)　**メソッド領域**
(2)　**ヒープ**
(3)　**Interpreter State**

メソッド領域においては，コンスタント・プールがクラスを問わず一元化されている点が異なる．この一元化は Dalvik VM における実行バイナリ，dex ファイルを作成する段階で行われる．

Runtime Data Area に替わり，Dalvik VM には Interpreter State と呼ばれる領域がある．スレッドごとに独立して存在する点は Runtime Data Area と同様である．また，内部にコンスタント・プールへのポインタ，PC，コール・スタックを持つ点も同様であるが，その構成は若干異なる．コール・スタックに pop，push される，フレームは次の要素からなる．

オブジェクトへのポインタ　コール・スタックの属する，インスタンスへの参照が格納される．

Dalvik レジスタ　メソッド内で利用される Dalvik レジスタ．メソッド単位で，コンパイ

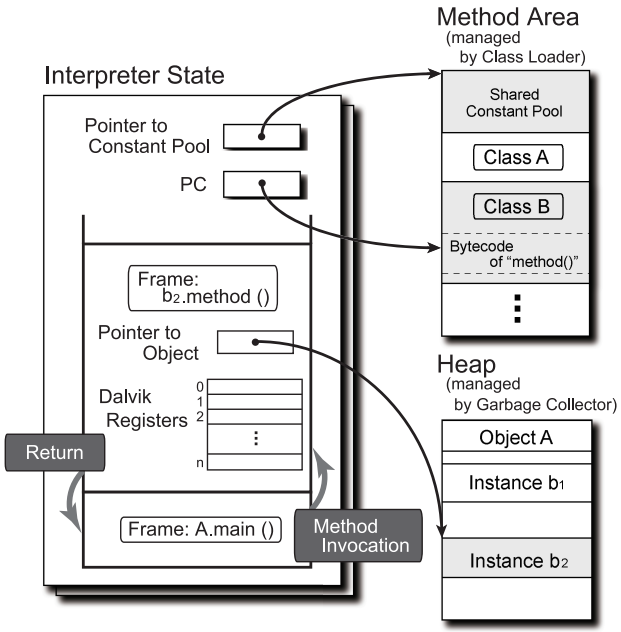

図 4　Dalvik インタプリタによる Dalvik バイトコード実行
Fig. 4　Dalvik bytecode execution by Dalvik VM interpreter.

ル時に決められた本数のレジスタを持つ．

Runtime Data Area ではコール・スタックごとに指定していたコンスタント・プールは，一元化されたことで Interpreter State で 1 つ有する形式となっている．

3.4　Java バイトコードとの比較

Dalvik VM が独自のバイトコードを採用した意図として，スタック・マシンに比べ，インタプリタの命令ディスパッチとメモリ・アクセスの削減をあげている．そして，実行バイナリの dex ファイルは前述のコンスタント・プールの一元化により，サイズの縮小を実現している．

Java VM の実行バイナリであるクラス・ファイル（.class）は，クラスごとにファイルが生成され，そこにクラスごとにバイトコードやコンスタント・プールが格納される．そして複数のクラス・ファイルが jar ファイルにまとめられる．

それに対し，dex ファイルは複数のクラスを 1 ファイルに持ち，重複する項目を消すことで，メソッド領域におけるコンスタント・プールの占める領域の節約を図っている．Android の Java で記述された共有ライブラリにおいて，dex ファイルは 10.3 MB のサイズとなる．それに対し，非圧縮の jar ファイルでは 21.4 MB，圧縮が有効な jar ファイルでは 10.6 MB のサイズとなる．圧縮されたクラス・ファイルよりもサイズが縮小することから，dex ファイルは効率の良いクラス・ファイルの格納を行っていることが分かる[2]．

4.　Dalvik アクセラレータ

我々が提案する Dalvik アクセラレータは Jazelle 方式と同様に，プロセッサのフェッチ，デコード・ステージの間に Dalvik バイトコードのデコーダを設ける．そしてフェッチ・ステージより入力された Dalvik バイトコードを逐次ネイティブ・コードへ変換することで，ハードウェア・アクセラレーションを実現する．

以下では，簡単のため，MIPS プロセッサ[11),12),27)] にアクセラレータを実装することを前提に説明する．ただし，以下で述べる方法の大部分は，アプリケーション・プロセッサのアーキテクチャに依存するものではない．出力する命令列の違いはあるものの，デコーダによるバイトコードの変換方法，また，4.5 節で述べるメモリ・アクセスの削減方法は，ARM プロセッサにアクセラレータを実装する場合も適用できる．むしろ，後者の方法は，MIPS よりも ARM の方が効果が高い．詳しくは 5 章で評価する．

4.1　アクセラレータの全体構成

Dalvik アクセラレータのデコーダは，図 2 と同様，ネイティブ・コードのパイプラインと並列に設けられており，解釈する命令セットに応じてネイティブ・モードと Dalvik モードを切り替える．

4.1.1　Dalvik モードへの切替え

Dalvik モードへの遷移は，Jazelle DBX と同様，専用命令によって行う．このため，無条件分岐命令を拡張した JRCD（Jump Register and Change Dalvik bytecode mode）命令を実装する．JRCD 命令が実行されると，ステータス・レジスタが更新され，Dalvik デコーダの on/off を切り替える．

4.1.2　ネイティブ・モードへの戻り方

Dalvik モードからネイティブ・モードへ戻る命令に，JRCM（Jump Register and Change MIPS mode）を定義，MIPS 側のデコーダに実装する．JRCM 命令が実行されると，ステータス・レジスタの更新とオペランドで指定されたレジスタの値に PC を設定する．

JRCM 命令は Dalvik デコーダが使用する命令で，ユーザ・プログラムでは利用しない．例外が発生するか，デコーダが変換できない命令がフェッチされたとき，デコーダは JRCM 命令を発行する．なお，変換できない命令とは，2.2 節で述べた，Jazelle DBX が変換対象としない Java バイトコードと同様の機能を有する Dalvik バイトコードを指す．以下では，通常の例外に加え，変換できない命令がフェッチされたことも含めて，**例外**と呼ぶことにする．

Dalvik モード中に例外が発生すると，アクセラレータは，発生した例外の要因を，後述する所定のレジスタ（ESTAT）へと書き込む．そして，上述の JRCM によって，例外ハンドラへとジャンプする．なお，例外ハンドラ・アドレスは，後述する所定のレジスタ（EHND）に格納される．例外ハンドラは，例外の要因に応じて，VM によるバイトコード実行を行ったり，通常の例外処理を行ったりする．

4.1.3　レジスタの使用

2.2 節で述べたように，Jazelle DBX では，モードのスイッチングを高速に行うため，VM が予約済みの物理レジスタ（表 1 の r6〜r8，r15）は Java モード時には別の目的には使用せず，Jazelle DBX はこれらの物理レジスタを共用する．同様に，Dalvik アクセラレータにおいても，VM が予約済みの物理レジスタは Dalvik モード遷移時の保存の対象とはせず，共用する．MIPS アーキテクチャ用の Dalvik VM においては，以下のレジスタが予約されている．

PC（r16） Dalvik インタプリタの PC.

FP（r17） フレーム内の Dalvik レジスタを格納した領域の先頭を指すポインタ.

GLUE（r18） インタプリタ・ステートの先頭を指すポインタ．コンスタント・プールやヒープを参照する際に使用される．

INST（r20） メソッド領域内のバイトコードを格納した領域の先頭を指すポインタ.

また，Dalvik モード中に発生した例外を処理するために，以下の物理レジスタを予約する．

EHND（r19） Dalvik モード中に発生した例外を処理する例外ハンドラのアドレス.

EOBJ（r21） 例外を起こしたオブジェクト参照.

ESTAT（r22） 例外レジスタ．発生した例外の種類が格納される．

EPC（r23） 例外を起こしたバイトコードのアドレス.

これらのほかに，ゼロ・レジスタなど，MIPS の規約上使用できない物理レジスタがある．また，デコーダがバイトコードを MIPS の命令列に変換する際，一時的に使用するレジスタがある．これらを**表 2** にまとめる．表 2 のレジスタを除いたレジスタが，Dalvik モード時に自由に利用できる．Dalvik アクセラレータは，これらのレジスタを，最近アクセスされた Dalvik レジスタの値の格納先として使用する．詳しくは 4.5 節で述べる．

表 2　MIPS アーキテクチャにおける Dalvik モード時のレジスタ割当て
Table 2　Register mapping in Dalvik mode for MIPS architecture.

番号	名称	目的
r0	zero	常時 0（MIPS の仕様[12]）
r1	at	マクロ命令が一時的に使用
r14，r15	SCRH0/1	デコードした MIPS 命令が利用可能な一時領域
r16	PC	Dalvik バイトコードの PC
r17	FP	Dalvik レジスタポインタ
r18	GLUE	Dalvik VM 資源ポインタ
r19	EHND	VM のハンドラ・アドレス
r20	INST	バイトコードのベースアドレス
r21	EOBJ	VM へ戻る要因であるオブジェクト参照
r22	ESTAT	VM へ戻る要因を収めたステータス値
r23	EPC	VM へ戻る要因となったバイトコードの PC
r24，r25	SCRH2/3	デコードした MIPS 命令が利用可能な一時領域
r26〜r31	-	カーネル・MIPS プログラムが使用

4.2　Dalvik デコーダの構造

図 5 に Dalvik デコーダの構造を示す．以下は各ブロックの機能，動作である．

バイトコードバッファ　10 バイト幅のバッファ．フェッチ・ステージでは，空きがある限り，毎サイクル，（r16 ではない MIPS の）プログラム・カウンタが指す 32 ビットのデータをこのバッファにフェッチする．次に述べる命令ジェネレータは，このバッファからバイトコード 1 つを取り込み，処理する．

命令ジェネレータ　バイトコード 1 つを対応する MIPS 命令列へと変換するユニット．変換作業は，後述する，変換テーブル，命令テーブル，DRMT の 3 つのテーブルを用いて行う．これらのテーブルから読み出された MIPS 命令列のひな形と，バイトコードから得られたオペランドにより，MIPS 命令を生成し，後続のユニットへ逐次発行する．

変換テーブル　Dalvik バイトコードのオペコード（Op）でインデクシングされたテーブル．各エントリは，デコーダが変換可能か示すフラグ（E），対応する命令テーブルへのポインタ（Next #）を持つ．また，命令テーブルの参照による命令発行のレイテンシの増加を抑えるため，命令テーブル上の対応する命令列を先頭から数命令分キャッシュ（Inst）する．

命令テーブル　変換結果の MIPS 命令列を保持したテーブル．各エントリは，1 つの MIPS

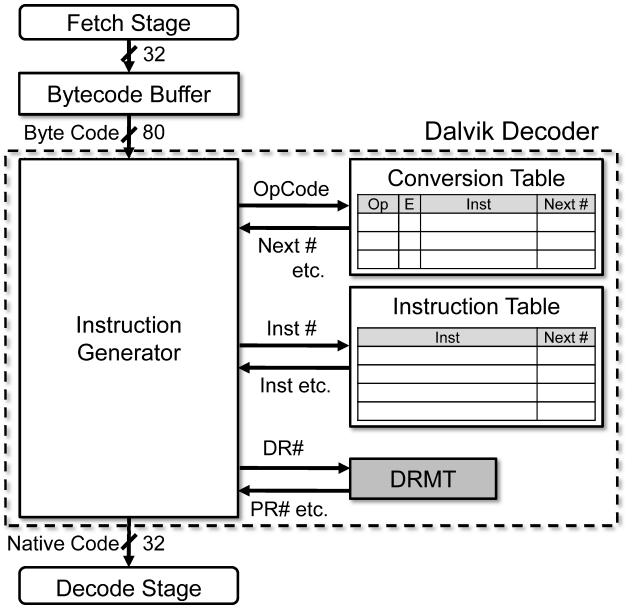

図 5　Dalvik デコーダのブロック図
Fig. 5　Block diagram of Dalvik decoder.

命令（Inst）と，次のエントリへのポインタ（Next #）を持つ．

DRMT　Dalvik Register Map Table．最近アクセスした Dalvik レジスタがどの物理レジスタにマップされているかを管理する表．後述するように，これによりロード/ストア命令を削減する．

4.3　ネイティブ・コードへの変換

命令ジェネレータが，1 つのバイトコードから MIPS 命令列へ変換する手順を示す．ここでは，Dalvik レジスタ間で加算演算を行う，`add-int v0, v1, v2` を例にする．Dalvik レジスタ 1 番と 2 番の間で加算，Dalvik レジスタ 0 番へ加算結果を書き込む．変換した結果より，次の 4 つの MIPS 命令列が出力される．ただし，$\$x$（$x = 2$，3，4，FP）は物理レジスタを表すものとする．特に，$FP は Dalvik レジスタポインタ（表 2 の r17）を表す．

(1)　`lw $2, 4($FP)`
(2)　`lw $3, 8($FP)`

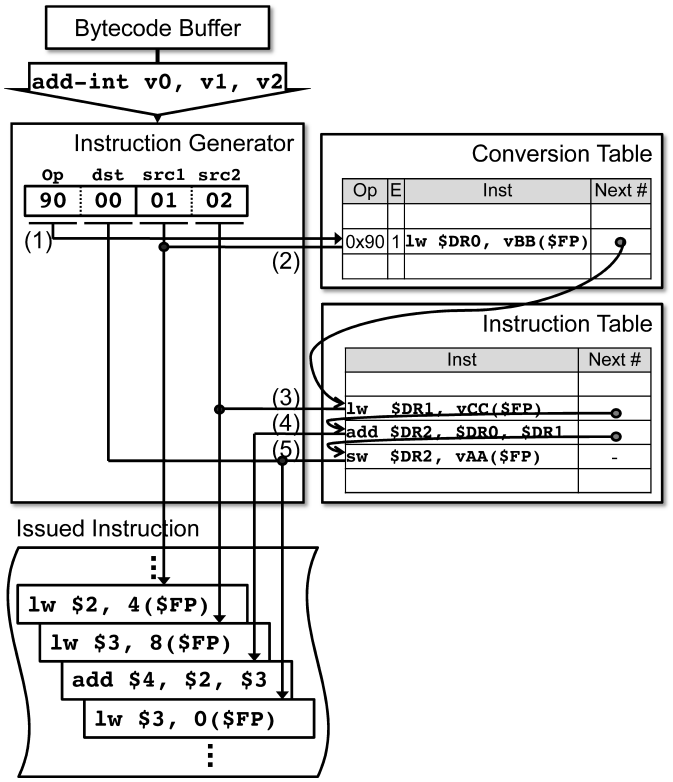

図 6　`add-int` 命令の変換例
Fig. 6　Example of `add-int` instruction conversion.

(3)　`add $4, $2, $3`
(4)　`sw $4, 0($FP)`

ここでは，MIPS レジスタ 2〜4 を読み出した Dalvik レジスタ，演算結果の保持に用いている．**図 6** に動作を示す．

(1)　バイトコードバッファから，フェッチした命令を取り出す．先頭から 1 バイト目のフィールドがオペコードを表す．`add-int` 命令のオペコードの値 0x90 で変換テーブルを参照する．

(2)　変換テーブルの 0x90 番目から，変換の可否，最初に出力する MIPS 命令のひな形，

次に参照すべき命令テーブルのポインタを取り出す．`add-int` 命令は変換できるので，ひな形を使い命令を出力する．ひな形を参照すると，第 2 オペランド（vBB）をある物理レジスタにロードしなければならないことが分かる．そこで，ジェネレータはバイトコードを参照し，第 2 オペランドが Dalvik レジスタ 1 番であることを取得する．ジェネレータは，Dalvik レジスタ 1 番のアドレス（4($FP)）を計算し，そこから値をロードする命令を発行する．なお，変換できない場合は，Dalvik モードからネイティブ・モードへ制御を戻す命令を発行する．

(3) 取得した命令テーブルのポインタより，後続する命令のひな形と，次に参照すべき命令テーブルのポインタを取得する．出力する命令は，第 3 オペランドで指定されている，Dalvik レジスタ 2 番より値をロードする `lw` 命令である．

(4) 繰り返し，次に参照する命令テーブルのポインタより，命令テーブルを参照し続ける．取り出したひな形より，ロードした Dalvik レジスタの値間で加算する `add` 命令を出力する．

(5) 繰り返し，加算結果を Dalvik レジスタ 0 番へストアする `sw` 命令を出力する．この命令テーブルの，次に参照する命令テーブルのポインタは空なので，命令の出力は終わる．次のバイトコードをフェッチする．

4.4 バイトコード単位でのロード/ストアによる効率の低下

前章で述べたように，Dalvik VM では，すべてのローカル変数，すなわち，Dalvik レジスタの値はメモリ上のある領域に存在する．そのため，ある Dalvik レジスタをソース/デスティネーションとする Dalvik バイトコードと等価な MIPS の命令列を生成する場合，通常，オペランドで指定された Dalvik レジスタ番号より Dalvik レジスタの位置するオフセット・アドレスを計算し，そこから値をロード/ストアする処理が行われる．このように，単純に Dalvik バイトコードを MIPS 命令列に変換すると，Dalvik レジスタに対するロード/ストア命令が多数発行される．

実際，Dalvik インタプリタ（MIPS アーキテクチャ版 Android 1.6 以後）は，上記のような命令を生成する．演算の前に，ソース Dalvik レジスタを一時的な情報を保持する物理レジスタへとロードする．この物理レジスタは，このバイトコードを処理している間のみ有効である．そして，デスティネーション Dalvik レジスタへ書き込む際は，そのつど，対応するアドレスへ演算結果をストアする．

Dalvik レジスタを物理レジスタへとマップし，Dalvik レジスタの読み出しは物理レジスタの読み出しに，Dalvik レジスタへの書き込みは物理レジスタへの書き込みに置き換えれば，不要なロード/ストアを大幅に削減できる．

Dalvik レジスタの値を格納するため，Dalvik レジスタと 1 対 1 に対応する，専用のハードウェア・レジスタを設けることも考えられる．しかし，3.2 節で述べたように，Dalvik レジスタは，仕様上は 65,536 個まで定義できる．そのような巨大なレジスタ・ファイルを設けるのは現実的でない．マッピングは，既存の物理レジスタに対して行った方がよい．

Dalvik レジスタの物理レジスタへのマッピングは，静的に行うことも考えられる．すなわち，表 2 において未使用の物理レジスタ r2〜r13 を Dalvik レジスタの 0〜11 に順に割り当てる．このようにすれば，割り当てられなかった Dalvik レジスタ（12 番以降）に関してはそのつどロード/ストアが必要となるが，割り当てられたものに関してはそのつどロード/ストアする必要がなくなる．次節で述べるテーブルも必要ない．

上述のように，MIPS の場合は 12 個の物理レジスタをマッピングに使用することができる．また，3.2 節で述べたように，大半のメソッドは Dalvik レジスタの使用個数が 12 個以内である．そのため，Dalvik アクセラレータを MIPS プロセッサに実装する場合は，静的マッピングでも十分な可能性が高い．

しかし，静的マッピングは，メソッド内で使用する Dalvik レジスタの数が，割当て可能な物理レジスタの数よりも多い場合には問題となる．マッピングできなかった Dalvik レジスタについては，上述のように，参照のたびにロード/ストアが必要となる．Dalvik バイトコードにおいてつねに，先頭から数個分の Dalvik レジスタが最も使用頻度が高いとは限らない．場合によっては，番号の大きい（12 番以降の）Dalvik レジスタの使用頻度が最も高い，ということもある．そのような場合に静的マッピングは効果的でない．

MIPS の場合は，Dalvik レジスタ数がマップ用の物理レジスタ数を下回るメソッドが大半であるが，そうでないメソッドも一部ある．また，携帯端末のアプリケーション・プロセッサとしては MIPS よりも ARM の方が一般的であるが，ARM の総物理レジスタ数は 16 本と少ない．したがって，Dalvik レジスタのマッピングに使用できる物理レジスタ数も，MIPS の場合よりも制限されることになる．このように，アクセラレータを実際のアプリケーション・プロセッサに実装することも想定し，マッピング方法を考える必要がある．

4.5 Dalvik Register Map Table

我々は，物理レジスタにロードされている Dalvik レジスタの情報を記憶するための，**Dalvik Register Map Table（DRMT）**を提案する．**図 7** に DRMT の構造を示す．今回アクセラレータの実装を進めている MIPS アーキテクチャは，32 個のレジスタを持つ．このうち前述の OS，アプリケーションのために保持するレジスタを除くと，12 個のレジス

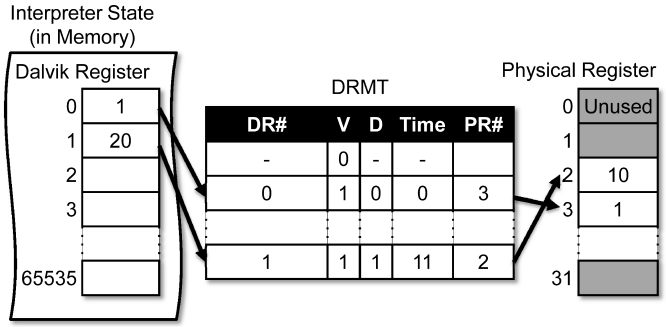

図 7　DRMT（Dalvik Register Map Table）
Fig. 7　DRMT (Dalvik Register Map Table).

タがアクセラレータで自由に利用できる．これらのレジスタを DRMT で管理することで，同時に最大 12 個の Dalvik レジスタを物理レジスタにロードできるようにする．

DRMT は 1 エントリが 1 つの物理レジスタに対応する．以下に DRMT の持つ要素を示す．

Dalvik レジスタ番号　その物理レジスタに割当て中の Dalvik レジスタ番号．命令ジェネレータが DRMT を検索するのに用いるタグである．

Valid ビット　エントリが有効か否かを示す 1 ビットのフラグ．

Dirty ビット　物理レジスタの値が dirty か否かを示す 1 ビットのフラグ．物理レジスタの値が，Dalvik レジスタより新しい場合にオンとなる．

タイムスタンプ　その Dalvik レジスタに最後にアクセスした時刻を記録する．新たに Dalvik レジスタをロードする際，未使用のレジスタがないこともある．そのような場合は，後述するように，最も最近利用されていない物理レジスタの値をメモリへ書き戻し，そこへ値をロードする．

物理レジスタ番号　Dalvik レジスタに対応する物理レジスタ番号を指す．

4.5.1　DRMT の動作

図 8 に，`add-int`，`sub-int` 命令がデコードされる過程を例に，DRMT の動作を示す．1 行は 1 サイクルに対応しており，左から，デコーダが出力する MIPS 命令，DRMT の状態，DRMT を参照した結果発行されるロード/ストア命令を示す．

ここでは説明を簡単にするため，DRMT エントリを 2 つとし，事前に一方のエントリに Dalvik レジスタ 2 番のマップ情報が収められているものとする．`add-int` 命令の変換結果，

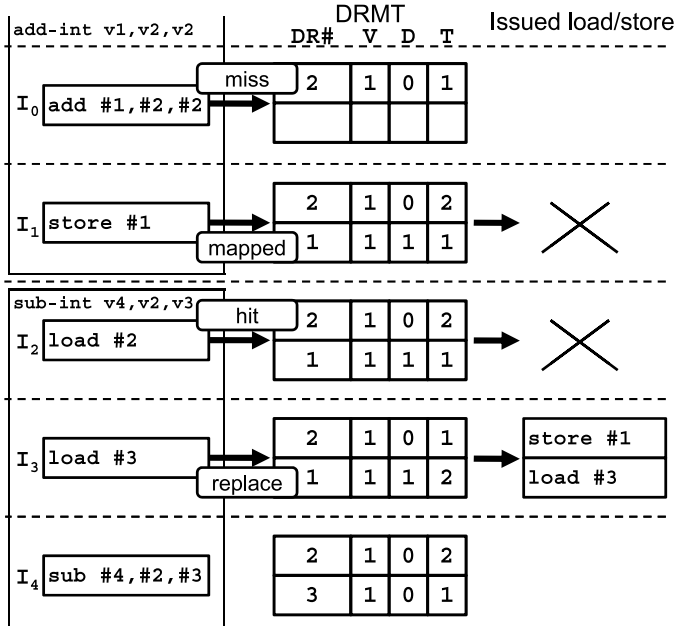

図 8　DRMT の動作例
Fig. 8　Example of DRMT operations.

MIPS 命令の `add` 命令を I_0 とする．以後，変換後の命令を I_1〜I_4 とし，Dalvik レジスタをシャープで始まる番号（#1〜3）で表す．

I_0：ミスする場合

I_0 は `add-int` 命令から変換された 3 番目の命令で，読み出した Dalvik レジスタより，加算を行う．演算結果が含まれる Dalvik レジスタの実体へストアを行うのは，I_1 ではあるが，宛先となる Dalvik レジスタが必要になった時点で，DRMT を参照する．ここでは演算結果の格納先となる #1 が DRMT マップされているか確認する．

DRMT には #1 に対応するエントリが存在しないので，新たに #1 と物理レジスタの対応をマップする．そして `add` 命令の宛先オペランドに，対応付けた物理レジスタがセットされる．`add` 命令により，Dalvik レジスタに対応する物理レジスタの値が更新されるので，該当する DRMT エントリの Time を更新して，Dirty ビットをオンにする．

I_1：ストア命令の省略

Dalvik レジスタへのストア動作は，DRMT へのヒット・ミスを問わず，発行はされない．後述するように，DRMT よりリプレースされるとき，物理レジスタの値をストアする．すでにストア予定の #1 は DRMT にエントリされているため，DRMT の操作も行わない．

I_2：ヒットする場合

I_2 は後続する `sub-int` 命令変換後の 1 つ目の命令で，オペランドで指定された Dalvik レジスタを物理レジスタへロードする．DRMT を参照し，ロードする #2 を探す．

I_1 の発行判定後の DRMT より，#2 は DRMT にマップ，物理レジスタにロードされている．このことから，変換した I_2 を省略できる．DRMT にヒットした場合は，該当する DRMT エントリの Time を更新する．

I_3：エントリのリプレースが発生

I_3 は `sub-int` 命令を変換した 2 つ目の命令である．オペランドで指定された Dalvik レジスタを物理レジスタへロードする．DRMT を参照し，ロードする #3 を探す．

I_2 の発行判定後の DRMT を参照すると，#3 は DRMT，物理レジスタに存在せず，物理レジスタにロードし DRMT へエントリを追加する必要がある．しかしすべての DRMT エントリが使用されている．この場合，LRU 方式でリプレースを行う．DRMT エントリ中の Time を参照し，最も最近利用されていない #1 をリプレースする．リプレースされる DRMT エントリの Dirty ビットがオンならば，物理レジスタの値を Dalvik レジスタへ反映させるために，ストア命令を発行する．そして #3 をロードする I_3 の発行と DRMT の更新を行う．

このように DRMT は，物理レジスタと Dalvik レジスタの対応関係を複数記憶することで，バイトコードを変換する際に生ずる，余分なロード/ストア命令を削減する．

なお，Java アクセラレータの 1 つである picoJava においては，オペランドに対する冗長な処理を削減する技術として，命令畳み込み[20] が提案されている．DRMT もオペランドに対する冗長な処理を省いており，一見すると，命令畳み込みと同様の技術であると思われるかもしれない．しかし，命令畳み込みと DRMT とは，以下で述べるようにまったく異なる．

picoJava には，ローカル変数とオペランド・スタックの一部を保持する，スタック・キャッシュと呼ばれるレジスタ・ファイルが存在する．メソッドの invoke 時に，引数に相当するローカル変数が，メモリからスタック・キャッシュへロードされる．以後のローカル変数に対する操作は，このスタック・キャッシュに対する操作に代替される．

このようなハードウェア構成であるため，たとえば，Java バイトコードにおけるオペランド・スタックを用いた以下の左のバイトコード列は，右の RISC 命令列に等しい．ただし，ra，rb，rc はスタック・キャッシュ上のローカル変数を格納したレジスタ，rt，rt' はスタック・キャッシュ上のオペランド・スタックの先頭に相当する 2 つのレジスタを表す．

```
iload a    →    mov ra rt
iload b    →    mov rb rt'
iadd       →    add rt rt rt'
istore c   →    mov rt rc
```

右のコード列に含まれる 3 つのレジスタ移動は，明らかに冗長である．命令畳み込みはこのような一連のバイトコード列を検出し，この冗長なレジスタ移動に相当する処理を削減する[*1]．すなわち，上述の 4 つの RISC 命令に代わり，「`add rc ra rb`」の 1 命令分の処理を行う．一方，DRMT は，限られた数の物理レジスタをローカル変数のキャッシュとして使用することで，ローカル変数に対する冗長なメモリ・アクセスを削減するのである．

4.5.2　DRMT の無効化とそのオーバヘッド

Dalvik デコーダは，例外が発生した場合，JRCM 命令を発行し，ネイティブ・モードへスイッチする．この際，Dalvik レジスタのマッピングに使用していた物理レジスタの内容をメモリ（Dalvik レジスタ）へと書き戻し，DRMT のすべてのエントリを無効化する．

例外を検出すると，Dalvik デコーダはまず，DRMT をもとに，すべての Dirty な Dalvik レジスタに対するストア命令を発行する．これにより Dalvik レジスタのコヒーレンシが保たれる．次いで，例外情報を所定のレジスタへと書き込み，JRCM 命令を発行する．そして，ネイティブ・モードへ戻り，VM による処理が開始される．この時点で Dalvik レジスタには正しい値が反映されているため，VM は，Dalvik モード中にマッピングに使用した物理レジスタを自由に使用できる．

このように，ネイティブ・モード遷移時には複数のストア命令の発行をともなうが，それが性能に与える影響は軽微と考えられる．なぜなら，それらのストアの大半は，（JIT や AOT などの）コンパイラによっても回避できないと考えられるからである．

Dalvik レジスタに対するロード/ストアが最も少ないコードは，Dalvik レジスタが最初に出現した際にそれを物理レジスタへとロードし，Dalvik レジスタの寿命が尽きたときに（それが Dirty であれば）ストアするコードだろう．Dalvik レジスタのロードは最初に 1 回

⋆1 一部の文献では，ローカル変数がメモリ上に存在し，iload/istore は RISC のロード/ストアと等価であると説明している．そのうえで，命令畳み込みは，そのようなローカル変数に対するロード/ストアを削減する技術であるかのように説明されているが，picoJava の whitepaper を見る限り，これらの説明は誤りだと思われる[21]．

行えばよく，以後その Dalvik レジスタを参照する命令は物理レジスタを参照すればよい．また，値が更新されるたびにストアする必要もない．演算途中では物理レジスタ上の値を更新し，該当する Dalvik レジスタの寿命が尽きたときにストアするのが最も効率が良い．

前述のように，Dalvik レジスタはローカル変数である．そのため，すべての Dalvik レジスタは，メソッドが終了するとき，あるいは，メソッド中で別の関数（Java のメソッドに限らない．C ライブラリの API を含む）を呼び出したとき，いったん寿命が尽きることになる．そのような場合には，たとえネイティブ・コードで記述されたプログラムであっても，使用した物理レジスタの値をメモリに退避する処理が基本的には必要である．

単純な関数コールであれば，caller 側と callee 側とで同じレジスタを使用することで，caller 側での Dirty な引数に対するストア，および，callee 側での引数に対するロードは省略できる．しかし，Java のメソッド呼び出しにおいては，名前解決のために，クラス・ローダなどのインタプリタ外部のソフトウェア・モジュールの呼び出しが行われる．すなわち，Java メソッドから Java メソッドを呼び出す処理は，ネイティブ・コード・レベルでは，クラス名やメソッド名を引数として前者の関数から名前解決のための関数をサブルーチン・コールし，後者の関数のアドレスを取得した後でそこへジャンプする，という処理に相当する．名前解決のための関数の中では，MIPS レジスタの利用規約に基づき，一時レジスタに割り当てられた Dalvik レジスタの値が破壊されることになる．

したがって，AOT コンパイラであっても，一時レジスタに割り当てた Dalvik レジスタは，すべて caller 側のメソッドで退避しなければならない．また，引数として callee 側のメソッドに渡す値も，上述のように，間に別のサブルーチン・コールが存在することから，いったんメモリに退避する必要がある．詳細は紙面の都合により省略するが，変換できないバイトコードのほとんどは，（バイトコードのメソッドではない）外部の関数呼び出しをともなうものである．DRMT 無効化時に発生する Dirty な Dalvik レジスタに対するストアは，このような処理と等価であると考えることができる[⋆1]．

5. 評　　価

DRMT の効果を Dalvik デコーダのシミュレータを用いて評価した．以下，詳しく述べる．

⋆1 AOT において，MIPS の r16〜r23（callee 側での退避が義務づけられたレジスタ）に割り当てられた Dalvik レジスタに関してはこの限りでない．そのため，厳密には，アクセラレーション中に invoke が発生した方が余分にストアを行う可能性がある．この余分なストアが性能に与える影響は今後詳細に評価する．

5.1 評 価 環 境

DRMT を用いた場合，Dalvik レジスタを静的にマッピングした場合，Dalvik レジスタが出現するたびにロード/ストアを行った場合とで，Dalvik デコーダが生成する命令数にどの程度の違いがあるかを評価した．前章で述べたように，DRMT によって，最近アクセスされた Dalvik レジスタについては，ロード/ストア命令の発行を回避できる．また DRMT は，単純な物理レジスタと Dalvik レジスタの対応付けに比べても，より効率的なロード/ストアが可能とみられる．本稿では，何命令のロード/ストアの発行が抑止できたかを評価する．

ここで，DRMT を用いない場合にデコーダが生成する命令列は，Dalvik インタプリタを介した場合にプロセッサが実行する命令列のサブセットとなる点に注意されたい．Dalvik インタプリタは，1 つのバイトコードを処理する際，それを MIPS の命令列に変換する作業に加え，バイトコードをフェッチするといった処理も行っている．すなわち，プロセッサから見れば，そうした処理自体も MIPS の命令列として記述され，実行されることになる．一方，アクセラレータではフェッチはハードウェアが行うため，デコーダがバイトコード 1 つを変換した際に生成される MIPS 命令数は，つねに，インタプリタによって生成される MIPS 命令数より少なくなる．

評価には 4 つのプログラム（Sieve，Matrix，IrrM，Radix）を用いた．各プログラムは，初期化処理などを除く，メイン・ループの部分についてのみ評価した．なお，すべてのメイン・ループは，アクセラレータが解釈できない複雑なバイトコードを 1 つも含んでいない．すなわち，すべてのバイトコードはアクセラレート可能である．

各プログラムの内容とパラメータを**表 3** に示す．デコーダが発行する命令数が近くなるようパラメータを調整した．なお，各プログラムで使用される Dalvik レジスタ数は，IrrM を除き，すべて 12 個以内である．

評価は Dalvik デコーダのシミュレータを用いて行う．このシミュレータは，エミュレー

表 3　評価に用いたプログラムとパラメータ
Table 3　Used programs in evaluation and parameters.

プログラム	内容	パラメータ
Sieve	エラトステネスのふるい	探索する範囲は 1,200,000.
Matrix	正則行列の行列積	行列の大きさは 128.
IrrM	非正則行列の行列積	2 つの行列のサイズは 126×127，127×128.
Radix	基数ソート	基数は 2，要素数は 524,288，値の最大値は 8.

タ，DRMT のシミュレータ，および，バイトコードジェネレータからなる．エミュレータがバイトコード列の実行をトレースし，その際の DRMT の挙動をシミュレートする．ジェネレータは，DRMT の状態に応じて，各バイトコードをネイティブな命令列に変換する．

次節，および，次々節からは，Dalvik デコーダを MIPS プロセッサへ実装した場合の評価結果，および，ARM プロセッサへ実装した場合の結果について述べる．DRMT のエントリ数は，前者の評価では 12 とする．後者の評価では，Dalvik レジスタのマップ用に使用できる物理レジスタ数が現時点では不明なため，2〜6 まで変化させた．

5.2　MIPS プロセッサにおける DRMT の効果

5.2.1　Dalvik レジスタ数が少ないメソッドの場合

デコーダが発行した MIPS 命令の割合を**図 9** に示す．グラフの横軸はプログラム名および DRMT の構成，縦軸は発行された命令の割合を示す．各プログラムは，左より，静的マッピング，4 セット 3 ウェイ，2 セット 6 ウェイ，フルセットアソシアティブの DRMT 構成による結果を示す．グラフは 3 色に塗りわけられており，下から順に，DRMT によって発行を回避できたロード/ストア（cancelled mem. inst），回避できなかったロード/ストア（executed mem. inst），ロード/ストア以外の命令（others）を表す．グラフより，DRMT を用いない場合，いずれのプログラムも，総発行命令数に対し，発行されたロード/ストア命令の割合が 3〜4 割ある．その大半が，DRMT によって削減できる．特に，Sieve では，削減できた命令の割合が全体の 40%にも達する．DRMT の構成が，一般にヒット率が下がる 4 セット 3 ウェイであっても，削減効果は多少下回る程度となった．そのほかの DRMT 構成においては，静的マッピングをわずかに上回る結果となった．

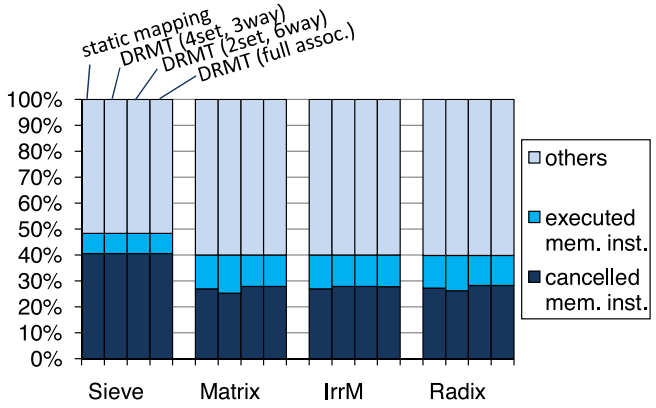

図 9　デコーダが発行した命令の内訳（MIPS で int 型のプログラムを実行した場合）
Fig. 9　The ratio of instructions issued by the decoder (in case where int type programs are executed on a MIPS processor).

図 10 に，DRMT を用いない場合に発行されたロード/ストア命令数，および，DRMT を用いた場合のロード/ストア命令数を示す．プログラムごとに並んだ 2 本の棒グラフは，左がロード，右がストアである．DRMT の構成は 4 セット 3 ウェイである．グラフより，ロードについては，いずれのプログラムも 60%以上の削減効果を示している．特に配列アクセス回数が少ない Sieve では，87%と高い削減効果を示した．ストアについては，約 65〜90%の削減効果を示した．トータルでは，DRMT を用いない場合と比べ，半分以上のロード/ストアを削減できることが分かる．

なお，削減できなかったロード/ストアの大半は，ヒープ上のデータに対するアクセスである．今回評価に用いたプログラムの多くは，配列を多用している．配列は Dalvik レジスタに参照のみ持ち，実体はヒープに存在するため，配列の実体に対するロード/ストアは，DRMT では回避できない．

5.2.2　Dalvik レジスタ数が多いメソッドの場合

前項の評価では，評価に用いたメソッドの Dalvik レジスタ数が少ないため，DRMT を用いた場合と静的マッピングの場合とで顕著な差が見られなかった．そこで，より多くの Dalvik レジスタを使用するメソッドを用いて評価する．

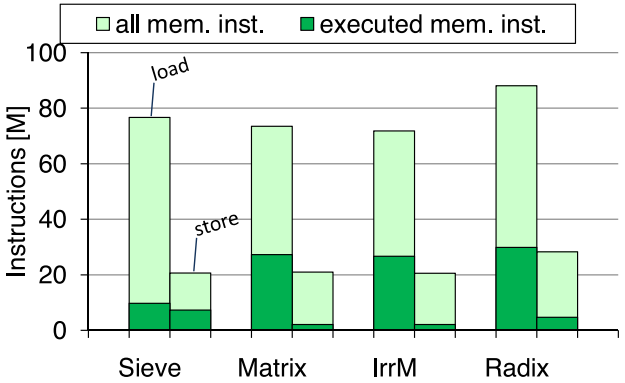

図 10　発行されたメモリ・アクセス命令数（左：ロード，右：ストア）
Fig. 10　Issued memory access count (left: load, right: store).

具体的には，先の評価に用いたプログラム中で使用されている変数の型を int 型から long 型へと変換し，変換したプログラムをベンチマークとして使用する．Dalvik VM の long 型は 2 つの連続した Dalvik レジスタを 1 つのレジスタとして扱うことで実現している．この変換の結果，各プログラムの Dalvik レジスタ数は，Sieve で 15 個，Matrix で 19 個，IrrM で 19 個，Radix で 20 個となった．

long 型への変換を行ったプログラムに対し，デコーダが発行した MIPS 命令の割合を**図 11** に示す．グラフの見方は図 9 と同様である．

総発行命令数に対し，発行されたロード/ストア命令の割合はいずれも 5 割を占めた．うち，削減した命令は 25〜40%となった．特に Sieve では DRMT 構成にかかわらず，全命令のうち 4 割以上のメモリ・アクセスを削減できている．

DRMT 構成の中で，Matrix，IrrM，Radix に共通して，4 セット 3 ウェイより，2 セット 6 ウェイの方が削減できた命令数が下回っている．これは，long 型の値を格納している Dalvik レジスタのうち，一方にアクセスが集中したためである．

配列へアクセスする `aget-`/`aput-` 系の命令のうち，アクセスする要素番号を示す第 3 オペランドは int 型でなければならない．いったん `long-to-int` 命令を介して型変換を行うが，この命令はペアとなっているレジスタから，下位側レジスタのみ移動させる動作に等しい．このような Dalvik レジスタへのアクセスにより，上記の 3 プログラムにおいて偶数番号の Dalvik レジスタを保持するセット側にアクセスが集中した．よって各セットごとにアクセスが偏り，偶数番号の Dalvik レジスタが必要なタイミングで物理レジスタに保持されない結果となった．

一方，4 セット 3 ウェイ構成では，偶数レジスタへの局所的なアクセスが 2 つのセットに分散した．リプレースされる頻度が低下し，DRMT へのヒット率が向上しメモリ・アクセスがより削減された．

このように，マッピングに使用できる物理レジスタ数が Dalvik レジスタ数よりも多い場合は，静的マッピングよりも DRMT の方が効果が高い．特に Radix については，DRMT を用いた場合，静的マッピングに比べて 15%の命令を削減した．

5.3　ARM プロセッサにおける効果

実際のアプリケーションでは，long 型の変数が多用されるとは考えにくい．そこで，物理レジスタ数が Dalvik レジスタ数よりも少ない環境として，ARM プロセッサに Dalvik デコーダを実装することを想定した評価を行う．ARM アーキテクチャは 16 本しか物理レジスタがないため，MIPS アーキテクチャに比べて，Dalvik レジスタのマッピングに利用できる物理レジスタ数が減ることになる．評価には 5.2.1 項で用いたプログラムを使用する．DRMT はフルアソシアティブとした．

デコーダが発行した ARM 命令の割合を**図 12** に示す．グラフの横軸は Dalvik レジスタ

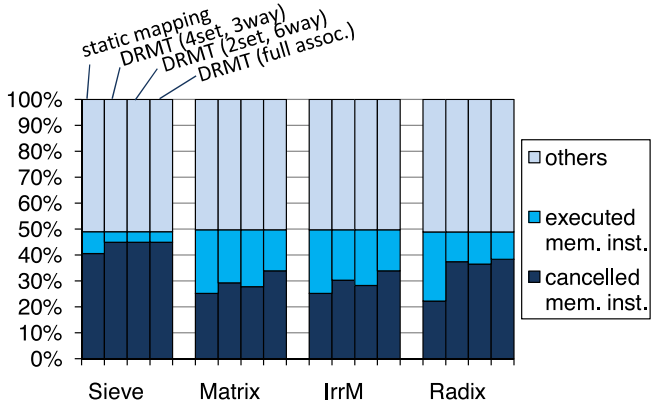

図 11　デコーダが発行した命令の内訳（MIPS で long 型のプログラムを実行した場合）
Fig. 11　The ratio of instructions issued by the decoder (in case where long type programs are executed on a MIPS processor).

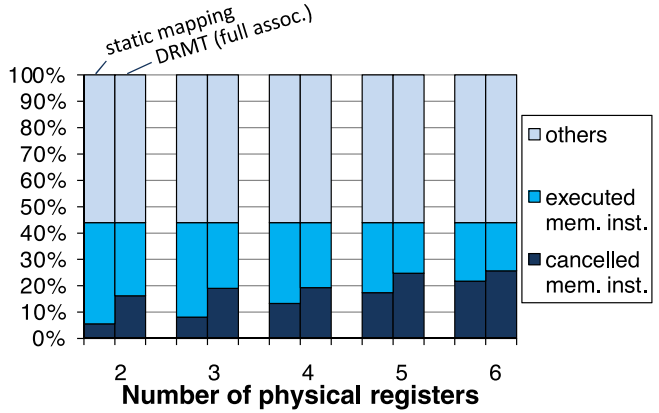

図 12　デコーダが発行した命令の内訳（ARM で int 型のプログラムを実行した場合）
Fig. 12　The ratio of instructions issued by the decoder (in case where int type programs are executed on an ARM processor).

に利用可能な物理レジスタの数である．各グラフの組は左が静的マッピング，右が DRMT フルアソシアティブ構成である．縦軸の割合は 4 つのベンチマークの命令比率の平均を示す．その他の命令は平均 56%存在し，残りの 44%がメモリアクセス命令であった．

グラフより，使用できる物理レジスタの個数が少ないと，DRMT の効果がより顕著になる．物理レジスタ 2 個のとき，静的マッピングが 5.6%の命令削減であるのに対し，2 エントリの DRMT は 16.2%の命令を削減した．Jazelle DBX と同様，4 つの物理レジスタをオペランドのマッピングに使用できた（表 1）と仮定した場合でも，静的マッピングが 13.3%だったのに対し DRMT が 19.3%と，命令削減率を増やすことができた．

5.4　デコーダが出力する命令列とコンパイラが生成する命令列との比較

DRMT を用いた場合，Dalvik デコーダが出力する命令列は，JIT や AOT などのコンパイラによって生成される命令列にかなり近いと予想される．その一方で，メモリ上に保持されるのはバイトコードであるため，ネイティブ・コードを保持する場合と比べてメモリを圧迫しない．

以下では，デコーダが出力する MIPS 命令列を，Android 2.2 より実装された Dalvik VM の JIT が生成する命令列と比較する．Dalvik VM の JIT は，トレース・ベースの JIT であり，トレースが一定回数実行されると，別スレッドによってコンパイルされたネイティブ・コードを実行する．JIT が生成したネイティブ・コードやコンパイル対象としたトレースなどの情報は，dalvikvm コマンドの `-Xjitverbose` オプションを用いて収集した．

現在の Dalvik VM の JIT は，重複したトレースに対して別のネイティブ・コードを生成することがある．そのため，JIT が生成したネイティブ・コード量を単純に合算すると，もとのバイトコードの 19 倍にもなってしまう．そこで以下では，JIT が生成した最長の命令列に着目し，それとデコーダが出力する命令列とを比較する．

5.4.1　デコーダが出力するコード

5.2.1 項で使用した Sieve プログラムのバイトコード列を**表 4** に示す．この区間のバイトコードは 4 命令からなり，バイトコードのサイズは 12 バイトである[*1]．

各バイトコードが 1 回ずつ実行されたとすると，DRMT を用いない場合，デコーダは 23 個（92 バイト分）の MIPS 命令を出力する（表の「DRMT 無効」の列）．これに対し，DRMT によってすべての Dalvik レジスタが物理レジスタにマッピングできたとすると，出力される命令数は 15 命令（60 バイト分）にまで縮小する．

表 4　評価コード Sieve からデコーダが出力する MIPS 命令のサイズ
Table 4　MIPS instruction size of sieve evaluation bytecode from Dalvik decoder.

		DRMT		JIT	
アドレス	バイトコード	無効	全ヒット	元コード	最適化後
	JIT プロローグ	-	-	4	0
001c	`if-ge v3, v5, 0022 // +0006`	20	12	36	28
001e	`aput-boolean v4, v0, v3`	52	40	48	36
0020	`add-int/2addr v3, v2`	16	4	12	4
0021	`goto 001c // -0005`	4	4	0	0
	例外ハンドリング	-	-	20	20
	JIT エピローグ（Chaining Cell など）	-	-	16	0
	合計	92	60	136	88

`add-int` 命令をはじめとする整数演算では，オペランドの Dalvik レジスタが DRMT にヒットした場合 `add` 命令 1 命令のみ出力される．MIPS の `add` 命令 1 命令は，もとのバイトコードと同じ 4 バイトである．したがって，これらの命令に関しては，バイトコードでもネイティブ・コードでも使用するメモリ量は変わらない．

ネイティブ・コードが使用するメモリ量がバイトコードに比べて多いのは，バイトコードの方は，配列のアクセスなどの複雑な処理を 1 つの命令（4 バイト）で表しているためである．たとえば，boolean 型の配列へ書き込む `aput-boolean` 命令は 10 個（40 バイト分）の MIPS 命令を出力する．これは，ストアするアドレスの計算やストアそのものに加え，ArrayIndexOutOfBoundsException などの例外処理のために，アクセスされた要素が配列のサイズを上回っていないか，配列の参照が null でないかをチェックしているためである．

5.4.2　JIT が生成するコード

JIT が生成するコードも，主要な部分については，デコーダが出力するコードと同様である．`add-int` 命令のような命令は，MIPS の `add` 命令 1 命令へとコンパイルされる．

また，Dalvik レジスタのロードは，各々1 回だけ行うように最適化されている．複数回使用されるものについてはコードの先頭でまとめてロードし，そうでないものは使用される直前でロードする．一方，Dalvik レジスタへのストアは，DRMT を用いた場合とは異なり，値が更新されるたびに行われている．アドレス 0x0020 の `add-int` 命令が 3 命令（12 バイト分）に変換されているのは，このような理由による．

また，JIT が生成するコードには，ネイティブ・コードによる実行を開始するためのプロローグ，インタプリタ実行へ戻るためのエピローグが含まれる．エピローグは，Dalvik VM

⋆1 左列のアドレスの単位は Dalvik VM のバイトコードの長さ単位，コードユニットである．1 コードユニットにつき 2 バイトとなる．

の JIT では Chaining Cell と呼ばれ，大きな割合を占めている．

Chaining Cell は，VM のハンドラのアドレスをレジスタへセットし，そこへジャンプするなどの処理を行っている．そのため，Chaining Cell 1 つ 1 つは 4 命令（16 バイト）とそれほど大きくはない．しかし，VM に制御を移した後，最初に実行されるバイトコードの PC を識別するため，（JIT 対象のバイトコードが分岐で終わるなど）ネイティブ・コードの脱出先が複数ある場合は，その分の Chaining Cell が必要となる．

また，例外処理に関して，JIT が生成するコードは DRMT が生成するコードよりも大きくなる．JIT が生成するコードでは，

(1)　例外の発生を判定，
(2)　生成した JIT コード中の例外ルーチンへジャンプ，
(3)　VM へ受け渡す，例外要因の定数をセット，
(4)　VM ハンドラ・アドレスの読み出し，
(5)　VM ハンドラへジャンプ，

の手順をふまえ，例外処理を開始する．一方，デコーダの例外判定は，

(1)　発生する可能性のある例外要因の定数をレジスタ $ESTAT にセット，
(2)　例外の発生を判定，
(3)　JRCM 命令を発行，VM ハンドラ・アドレスへジャンプしつつネイティブ・モードへ切り替える，

となる．

デコーダの出力するコードに比べ，JIT コードは，自身に含まれる例外ルーチンへのジャンプと VM ハンドラへのアドレスの取得という動作が増えている．デコーダでは VM のハンドラアドレスは，アクセラレーション開始前に $EHND にセットされている．よって JRCM 命令でレジスタ参照ジャンプをしつつモードを戻すだけでよい．

このような理由により，JIT が生成するコードは，元のバイトコードはおろか，DRMT が出力するコードよりも膨張してしまう．表 4 より，JIT が生成するコードのメモリ量はバイトコードのメモリ量と比べて 11.3 倍にもなることが分かる．プロローグや Chaining Cell，例外判定部分を除き，主要な部分のみのコード量を取り出してみても，DRMT を用いた場合とほとんど変わらない．このように最適化されたとしても，元のバイトコードに比べて 7.3 倍に膨張してしまう計算である．

生成するネイティブ・コードをホット・スポットに限定すれば，コード全体の膨張率を抑えることはできよう．しかし，コードの膨張率と速度とのバランスを考えながら，チューニングを行うのは労力を要する．また，JIT を使用するためには，トレースを検出するための記憶領域なども必要である．以上の理由により，メモリに対する負荷という点では，JIT や AOT よりもハードウェア・アクセラレータの方が優れている．

6. まとめと今後の課題

本稿では，Dalvik バイトコードをハードウェアで実行する，Dalvik アクセラレータを提案した．また，バイトコードを単純にネイティブ・コードに変換した場合にロード/ストア命令が多数発行されるという問題に対し，DRMT を用いる手法を提案した．評価の結果，MIPS アーキテクチャ上の 12 エントリ DRMT では，最大 90%のストアを削減でき，トータルでは約 30%の命令の発行を抑制できることが分かった．使用可能な物理レジスタが 4〜6 個と限られた状態では，DRMT は静的マッピングに比べよりロード/ストア命令の削減効果が見られた．

今後の課題を以下にあげる．

- DRMT が実行時間に与える影響を評価する．今回，命令の削減数についてのみ評価を行ったが，プログラムの実行時間は命令数に単純に比例するわけではない．命令数の削減が実行サイクル数に与える影響について評価する必要がある．
- 今回評価に用いたプログラムはすべて，Dalvik デコーダがデコードできる命令であった．しかし，実際のアプリケーションには，アクセラレーションできない命令もある．その場合，前述のように，実行モードを戻し，VM がバイトコードを実行する必要がある．したがって，実際には，DRMT による命令削減の効果は，今回示した結果よりも低いと予想される．今後は，VM で実行しなければならない命令を含んだプログラムを用いて，アクセラレートした場合としなかった場合とで，実行時間にどの程度の違いが現れるかを評価する．
- DRMT を含むデコーダを実際に実装し，回路規模がどの程度増加するかを評価する予定である．

謝辞　本研究の一部は文部科学省共生情報工学推進経費による．

参　考　文　献

1) ARM: *ARM Architecture Reference Manual* (2005).
2) Bornstein, D.: Dalvik VM internals, *Google I/O Developer Conference* (2008).
3) Capewell, P. and Watson, I.: A RISC Hardware Platform for Low Power Java,

International Conference on VLSI Design, pp.138–143 (2005).
4) Google, Inc.: Android. http://www.android.com/
5) Google, Inc.: Android Developers. http://developer.android.com/
6) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, chapter 3, Prentice Hall (1999).
7) Markus, L.: Java to Go: Part 1, *Microprocessor Report*, Vol.15, No.2 (2001).
8) Markus, L.: Java to Go: Part 4, *Microprocessor Report*, Vol.15, No.6 (2001).
9) Markus, L.: Java to Go: The Finale, *Microprocessor Report*, Vol.15, No.6 (2001).
10) McGhan, H. and O'connor, M.: PicoJava: A Direct Execution Engine for Java Bytecode, *IEEE Computer*, Vol.31, pp.22–30 (1998).
11) MIPS Technologies, Inc.: MIPSANDROID. http://mipsandroid.org/
12) MIPS Technologies, Inc.: *MIPS32 Architecture For Programmers Volume I: Introduction to the MIPS32 Architecture* (2001).
13) Mizuno, H., Irie, N., Uchiyama, K., Yanagisawa, Y., Yoshioka, S., Kawasaki, I. and Hattori, T.: SH-Mobile3: Application Processor for 3G Cellular Phones on a Low-Power SoC Design Platform, *Hot Chips 16* (2004).
14) Paleczny, M., Vick, C. and Click, C.: The Java Hotspot server compiler, *JVM'01: Proc. 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, USENIX Association (2001).
15) Porthouse, C.: *Jazelle DBX Technology: ARM Acceleration Technology for the Java Platform* (2005).
16) Proebsting, T.A., Townsend, G., Bridges, P., Hartman, J.H., Newsham, T. and Watterson, S.A.: Toba: Java for applications a way ahead of time (WAT) compiler, *COOTS'97: Proc. 3rd Conference on USENIX Conference on Object-Oriented Technologies* (*COOTS*), USENIX Association (1997).
17) Shi, Y., Gregg, D., Beatty, A. and Ertl, M.A.: Virtual machine showdown: Stack versus registers, *Proc. 1st ACM/USENIX International Conference on Virtual Execution Environments* (*VEE '05*), pp.153–163 (2005).
18) Steele, S.: *Accelerating to Meet the Challenge of Embedded Java* (2001).
19) Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H. and Nakatani, T.: A dynamic optimization framework for a Java just-in-time compiler, *Proc. 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.180–195, ACM (2001).
20) Sun Microsystems: picoJava-II Microarchitecture Guide (1999).
21) Sun Microsystems: picoJava-II Programmer's Reference Manual (1999).
22) 吉田昌平：プロローグ　注目を集める Android，その理由は　なぜ組み込み産業界は Android に注目しているのか（特集　Android が動作する Linux の移植から C 言語によるアプリ作成まで Android × Linux = 次世代組み込み開発），インターフェース，Vol.36, No.4, pp.44–47 (Apr. 2010).
23) 太田　淳，三輪　忍，中條拓伯：Dalvik アクセラレータ：Android 端末における Java アプリケーションの高速実行機構，組込みシステムシンポジウム（ESS2010），pp.13–22 (2010).
24) 太田　淳，茂手木貴彦，三輪　忍，中條拓伯：Dalvik アクセラレータのための MIPS シミュレータを用いた評価環境，先進的計算基盤システムシンポジウム（SACSIS2010），pp.113–114 (2010).
25) 中川輪土：Android 高速化テクニック，組み込みプレス，Vol.16, pp.8–14 (2009).
26) 中村成洋，相川　光：ガベージコレクションのアルゴリズムと実装，秀和システム (2010).
27) 渡邉伸平，藤枝直輝：MIPS システムシミュレータ SimMips を活用した組込みシステム開発の検討，情報処理学会研究報告，Vol.2008, No.116, pp.23–28 (2008).
28) 鷲見　豊（訳），Mayer, J. and Downing, T.（著）：JAVA バーチャルマシン，O'REILLY (1997).

(平成 22 年 9 月 30 日受付)
(平成 23 年 1 月 21 日採録)

太田　　淳（正会員）

1984 年生．2005 年育英工業高等専門学校情報工学科卒業．2007 年東京農工大学工学部情報コミュニケーション工学科卒業．2008 年同大学大学院工学府博士前期課程情報工学専攻修了．現在，同大学院工学府博士後期課程電子情報工学専攻に在籍．2007 年 4 月より，サレジオ工業高等専門学校情報工学科非常勤講師．組み込みシステムに関する研究に興味を持つ．組込みシステムシンポジウム 2010 優秀論文賞を受賞．

三輪　忍（正会員）

1977 年生．2000 年京都大学工学部情報学科卒業．2002 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．2005 年同大学院情報学研究科通信情報システム専攻博士後期課程学習認定退学．同年京都大学大学院法学研究科助手．2008 年 1 月より東京農工大学工学府特任助教，現在に至る．博士（情報学）．計算機アーキテクチャ，並列処理，組み込みシステムの研究に従事．組込みシステムシンポジウム 2010 優秀論文賞等を受賞．電子情報通信学会，人工知能学会各会員．

中條　拓伯（正会員）

1961 年生まれ．1985 年神戸大学工学部電気工学科卒業．1987 年同大学大学院工学研究科修了．1989 年同大学工学部助手の後，1998 年より 1 年間 Illinois 大学 Urbana-Champaign 校 Center for Supercomputing Research and Development（CSRD）にて Visiting Research Assistant Professor を経て，現在，東京農工大学大学院工学研究院准教授．プロセッサアーキテクチャ，並列処理，クラスタコンピューティング，リコンフィギャラブルコンピューティングに関する研究に従事．電子情報通信学会，IEEE CS，ACM 各会員．博士（工学）．