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

Dalvik Accelerator: a Hardware Mechanism to Accelerate
Java Applications on an Android Device

2013 年 7 月 29 日

太田 淳

Atsushi OHTA

東京農工大学大学院 工学府 電子情報工学専攻

2013 年度 博士論文

要旨

「Android」のアプリケーション実行基盤である「Dalvik VM」には，実行速度に課題がある．また Dalvik VM における高速化手法は乏しい現状にある．本論文では，Android アプリケーションの高速実行を実現する機構を提案する．1 つ目は Dalvik VM が実行するバイナリ「Dalvik バイトコード」を入力，プロセッサの命令セットに変換する「Dalvik バイトコード・アクセラレータ」である．2 つ目は，Dalvik バイトコードにおいて演算対象である Dalvik レジスタのロード・ストア削減機構「DRMT (Dalvik Register Map Table)」である．3 つ目は Dalvik バイトコード・アクセラレータと Dalvik VM 間のモード遷移コストを削減する機構である．この 3 つの機構により，VM での実行や JIT に比べ効率的な命令の生成・実行を示す.

第 1 章「緒言」では， Android の著しい市場拡大を示した．一方で，アプリケーションの実行環境である Dalvik VM の実行速度には課題があることを述べた．現在の Dalvik VM は Java VM に比べて高速化手法に乏しく，動作速度を向上させる意義があることを述べた．それを踏まえ，ハードウェアにより Android アプリケーションのバイナリである Dalvik バイトコードを直接実行する，Dalvik バイトコード・アクセラレータを提案，その有効性を示すことを本論文の目的として設定した.

第 2 章「Java VM」では，Java VM の動作について述べた．Java においてはすでに高速化手法が複数存在している．本論文で実装する手法を紹介する手前，Java VM の内部アーキテクチャ，動作原理について紹介した.

第 3 章「Java における高速化手法」では，既存の Java VM における高速化手法を述べた．ソフトウェアによる手法としては，JIT，AOT(Ahead Of Time) コンパイルを挙げた．いずれの方法も，生成したネイティブ・コードを保持するために使用する主記憶・補助記憶が増加することが課題である．ハードウェアによる手法は，プロセッサとの接続関係によって，コプロセッサ型，データ共有型，制御共有型の 3 種類に大分できる．各実装方式のうち制御共有型の「Jazelle DBX」が回路規模の目安となるゲート数が少なく，組み込み機器向けに適していることを挙げ．Dalvik バイトコード・アクセラレータの実装方式として用いることを述べた.

第 4 章「Dalvik アーキテクチャ」では，Dalvik VM の内部アーキテクチャ，動作，Dalvik バイトコードについて述べた．Dalvik VM はレジスタベースの VM であり，VM が実行するバイナリも Java とは異なる．章末にて Java VM との相違点を示し，Dalvik VM の高速化手法がまだ不十分であることを述べた.

第 5 章「Dalvik バイトコード・アクセラレータ」では，本論文で提案するアクセラレータの仕様と動作を述べた．まず，Jazelle DBX を基として，MIPS アーキテクチャのプロセッサ・パイプラインのフェッチ・ステージとデコード・ステージの間に，Dalvik デコーダを搭載することを示した．Dalvik デコーダは，Dalvik バイトコードをデコードしプロセッサのネイティブ・コードに変換する．続いて Dalvik デコーダの内部構造と，どのようにして Dalvik バイトコードを入力，変換し，ネイティブ・コードを生成する過程を述べた.

第 6 章「DRMT」では，Dalvik VM やバイトコード・アクセラレータの生成する命令には，メモリのロード・ストアの無駄が生じていることと，問題の解決手法として DRMT を用いる手法について述べた．Dalvik VM の演算対象である Dalvik レジスタは，メモリ上に存在する配列である．特に高速化を行っていない VM やクセラレータでは，バイトコード単位でメモリのロード・ストアが発生することとなる．DRMT により複

数の Dalvik バイトコードを跨いで，ロードした Dalvik レジスタを物理レジスタに保持する．そして DRMT により Dalvik レジスタのロード・ストアを削減する動作例を示した．

第 7 章「遷移コストの削減機構」では，Dalvik バイトコード・アクセラレータと Dalvik VM の間で実行モードの遷移が発生する際，大きなサイクルを要していることを示した．これは Dalvik バイトコード・アクセラレータの性能向上を損ねる．これに対し，アクセラレーション可能，不可能なバイトコードの連続を予測し，モード遷移を減らす，高速に各モードが用いるレジスタを切り替えるレジスタウィンドウを提案，評価した．

第 8 章「プロセッサシミュレータ上での Android 実行」では，Dalvik バイトコード・アクセラレータを実装し，性能を評価する環境として用いるプロセッサシミュレータ「SimMips」について，その概要とアクセラレータを搭載するのに必要な変更を述べた．変更を加えた SimMips 上でベンチマーク・プログラムを実行し，実際の携帯端末向けプロセッサに近い特性あることを示した．

第 9 章「評価」では，本論文で示したアクセラレータならびに DRMT について，高効率な命令生成を評価しその結果を述べた．評価は 2 種類行った．1 つ目は DRMT により Dalvik バイトコードから，不必要な Dalvik レジスタのロード・ストアが削減されているか評価した．2 つ目はアクセラレータが生成した命令について JIT と比較し，効率的な命令生成が行われているかを評価した．

第 10 章「結論」では，本論文の結論を述べた．まずアクセラレータならびに DRMT により，VM や JIT に比べより効率的な命令生成が行われる要因を述べた．次に今後の展望として，SimMips やハードウェアへのアクセラレータの実装の可能性と，それにより得られる評価について述べた．

目 次

要旨

目次 v

第 1 章 緒言 1

1.1 携帯電話を取り巻く環境 . . . 1
1.2 Android 普及の背景 . . . 1
1.2.1 スマートフォン，タブレットにおける普及 . . . 1
1.2.2 Android の利用用途の拡大 . . . 2
1.2.3 用途の拡大の背景 . . . 2
1.3 Android のアプリケーション・アーキテクチャ . . . 4
1.3.1 Java バイトコードに対する高速化手法 . . . 4
1.3.2 Dalvik バイトコードに対する高速化手法の現状 . . . 5
1.4 研究の目的 . . . 6
1.4.1 Dalvik デコーダの実装 . . . 6
1.4.2 DRMT 機構の効果の測定 . . . 6
1.4.3 遷移コストの削減 . . . 6
1.5 本研究が与える影響 . . . 7
1.6 論文の構成 . . . 7

第 2 章 Java Virtual Machine 9

2.1 Java VM による Java プログラムの実行 . . . 9
2.2 Java バイトコード . . . 9
2.3 Java バイトコード・インタプリタ . . . 10
2.3.1 メソッド領域 . . . 10
2.3.2 ヒープ . . . 11
2.3.3 Runtime Data Area . . . 11
2.4 全体の処理の流れ . . . 12
2.5 まとめ . . . 12

第 3 章　関連研究: Java における高速化手法 **13**
3.1 ソフトウェアによる方式 13
3.1.1 JIT コンパイル 13
3.1.2 AOT コンパイル 14
3.1.3 命令セットによるネイティブコード生成支援 14
3.2 ソフトウェアによるネイティブコード生成の課題 14
3.3 ハードウェアによる方式 15
3.3.1 JVXtreme 15
3.3.2 picoJava 16
3.3.3 JSTAR 17
3.3.4 BTU 17
3.3.5 Jazelle DBX 18
3.4 Java アクセラレータの比較 21
3.5 まとめ 22

第 4 章　Dalvik アーキテクチャ **23**
4.1 Dalvik VM 採用の背景 23
4.2 Dalvik バイトコードの生成 23
4.3 dex ファイル 24
4.4 Dalvik バイトコードの仕様 24
4.5 Dalvik レジスタ 25
4.6 Dalvik VM の内部構造 26
4.6.1 メソッド領域 27
4.6.2 ヒープ 27
4.6.3 Interpreter State 28
4.7 Java VM との比較 28
4.7.1 スタック・マシンとレジスタ・マシン 28
4.7.2 GC の方式 28
4.7.3 格納ファイルの比較 29
4.8 Android アプリケーションの動作 29
4.9 VM の高速化手法 30

第 5 章　Dalvik バイトコード・アクセラレータ **31**
5.1 Dalvik アクセラレータの実装方針 31
5.2 適合するアーキテクチャ 31
5.3 プロセッサ上での構成 32
5.4 モードの切替 32

5.4.1 Dalvik モードへの切替 33
5.4.2 ネイティブモードへの切替 33
5.5 MIPS アーキテクチャにおける実装例 34
5.6 レジスタの使用 34
5.7 アクセラレータの構造 36
5.7.1 バイトコードバッファ 36
5.7.2 命令ジェネレータ 38
5.7.3 変換テーブル 38
5.7.4 命令テーブル 39
5.7.5 DRMT 40
5.8 プロセッサに追加する命令 41
5.8.1 例外チェック命令 41
5.8.2 分岐用パイプライン制御命令 42
5.9 ネイティブコードへの変換 43
5.9.1 add-int 命令の変換 44
5.9.2 add-long/2addr 命令の変換 45
5.9.3 aput-int 命令の変換 47
5.10 まとめ 48

第 6 章 DRMT **49**
6.1 Dalvik レジスタの冗長なロード・ストアの存在 49
6.2 Dalvik レジスタのロード/ストアを削減する手法の考察 50
6.2.1 Dalvik レジスタを格納する専用のハードウェア・レジスタ 51
6.2.2 Dalvik レジスタの物理レジスタへの静的マッピング 51
6.3 物理レジスタと Dalvik レジスタのマップ 52
6.4 DRMT の動作 53
6.4.1 命令畳込みとの相違点 55
6.5 DRMT の無効化とそのオーバヘッド 56
6.6 ハードウェア上での実装方法 57

第 7 章 遷移コストの削減機構 **59**
7.1 実行モード遷移時のコストの大きさ 59
7.2 実行モード遷移コストの削減 60
7.2.1 Dalvik アクセラレータのモード遷移予測 60
7.2.2 2 つの実行モードに対応するレジスタウィンドウの設置 62
7.2.3 性能評価 64
7.3 まとめ 64

第 8 章　プロセッサシミュレータ上での Android の実行　66

8.1 SimMips の概要 66
8.2 SimMips のアーキテクチャ 67
8.2.1 命令の実行 67
8.2.2 メモリアクセス 68
8.2.3 周辺機能 69
8.3 SimMips の変更点 69
8.3.1 パイプライン化 69
8.3.2 メモリアクセスに伴うレイテンシとキャッシュのシミュレーション 70
8.3.3 未定義命令における例外発行 71
8.4 Android の起動環境の構築 71
8.4.1 カーネルの設定 71
8.4.2 シェルへのアクセス 72
8.5 評価 73
8.5.1 評価環境 73
8.5.2 ベンチマーク結果 74
8.6 まとめ 74

第 9 章　評価　76

9.1 DRMT によるロード・ストアの削減効果 76
9.1.1 評価環境 76
9.1.2 MIPS プロセッサにおける DRMT の効果 77
9.1.3 ARM プロセッサにおける効果 80
9.1.4 実行サイクル数を基準とした概算比較 81
9.2 デコーダが生成する命令列とコンパイラが生成する命令列との比較 82
9.2.1 デコーダが生成するコード 83
9.2.2 JIT が生成するコード 84

第 10 章 結論　86

10.1 研究成果 86
10.1.1 Dalvik アクセラレータの実装方法 86
10.1.2 Dalvik アクセラレータによる命令生成 86
10.1.3 DRMT によるデコード命令の削減 87
10.2 今後の課題 87
10.2.1 SimMips への Dalvik アクセラレータ実装と評価 87
10.2.2 ハードウェアでの実装 88
10.2.3 効率的な Dalvik レジスタのマッピング手法の考案 89

謝辞 **90**

参考文献 **94**

付 録 A　研究業績 **95**

A.1　論文誌 95
A.2　査読付き国際会議 95
A.3　査読付き国内会議 95
A.4　研究会，査読なしシンポジウム，技術報告書 96

図 目 次

1.1 Android のアーキテクチャ図 3

2.1 インタプリタによる Java バイトコード実行 11

3.1 Java アクセラレータの実装方式 15
3.2 命令畳み込みの例 17
3.3 JSTAR のブロック図 18
3.4 BTU のブロック図と CPU との接続関係 19
3.5 Jazelle 方式のブロック図 20

4.1 Dalvik バイトコードのフォーマット例 25
4.2 Android 標準アプリケーションにおけるメソッド毎の Dalvik レジスタ使用本数 26
4.3 Dalvik インタプリタによる Dalvik バイトコード実行 27

5.1 Dalvik デコーダを組み込んだプロセッサのブロック図 33
5.2 Dalvik デコーダのブロック図 37
5.3 バイトコードバッファ 積まれた状態 38
5.4 バイトコードバッファ バイトコードを形成 39
5.5 MIPS 命令セットのフォーマット 40
5.6 分岐命令の動作 43
5.7 add-int 命令の変換例 45
5.8 aput-int 命令の動作模式図 47

6.1 冗長なロード・ストアを含むバイトコードからデコードされた MIPS 命令列の組み合わせ . 50
6.2 理想的なバイトコードからデコードされた MIPS 命令列 51
6.3 DRMT (Dalvik Register Map Table) 53
6.4 DRMT の動作例 54

7.1 Dalvik アクセラレータの実行フローチャート 61
7.2 実行モード遷移を削減する Dalvik アクセラレータの実行フローチャート 62
7.3 レジスタウィンドウのアーキテクチャ 63
7.4 実行命令数とオーバヘッドの削減量 64

8.1 SimMips のプロセッサ情報と各ステージの関係 68
8.2 シングルサイクルモードとマルチサイクルモードのシミュレーション動作 68
8.3 SimMips におけるパイプライン実行 70
8.4 コンソールからの Dalvik VM 実行例 73
8.5 SimMips と Google Dev Phone 1 のベンチマークスコア 75

9.1 デコーダが生成した命令の内訳 （MIPS で int 型のプログラムを実行した場合） 78
9.2 生成されたメモリ・アクセス命令数（左:ロード 右:ストア） 79
9.3 デコーダが生成した命令の内訳（MIPS で long 型のプログラムを実行した場合） 80
9.4 デコーダが生成した命令の内訳（ARM で int 型のプログラムを実行した場合） 81
9.5 実行サイクル数の削減 （ARM で int 型の Sieve プログラムを実行した場合） 82

表 目 次

3.1 Jazelle DBX におけるレジスタ割り当て（一部） 21
3.2 主な Java アクセラレータの比較 22

4.1 dex ファイルと jar ファイルの効率比較 29

5.1 MIPS アーキテクチャにおける Dalvik モード時のレジスタ割り当て 35

6.1 MIPS，ARM 各アーキテクチャにおける DRMT のビットサイズ 57

7.1 8 つのレジスタの目的 63

8.1 評価環境の主なスペック 74

9.1 評価に用いたプログラムとパラメータ 77
9.2 評価コード Sieve からデコーダが生成する MIPS 命令のサイズ 83

第1章　緒言

1.1　携帯電話を取り巻く環境

携帯電話の普及は著しく続いている．日本においては携帯電話の IP 接続契約台数が 1 億台を超えている [1]．2012 年の全世界での販売台数は約 17 億台と巨大な市場となっている [2].

普及とともに，高性能化も著しい．かつて通話や SMS に代表されるテキストベースのメッセージ送受信が主であったが，データ通信による簡易なウェブサイトの閲覧，マルチメディアデータの送受信，そして利用者によるアプリケーションの追加・実行が可能と発展していった.

そして高機能化が進む中で**スマートフォン**と呼ばれる，汎用的なアプリケーションが実行でき，汎用的なウェブサイト閲覧が可能なインターネットへの接続機能を持つ携帯電話が登場した．スマートフォンに用いられるプラットフォームとして，Symbian，BlackBerry，Windows Mobile（現 Windows Phone），iPhone OS（現 iOS）などが発表されてきた．そのような中，2008 年に Google 社が携帯電話向けプラットフォーム，**Android** を発表した [3][4].

1.2　Android 普及の背景

Android は著しい普及が進み，携帯電話，タブレットデバイスでの普及に限らず，さまざまな用途への流用が広がっている [5]．まず以下にその背景とどのような用途への展開が進んでいるかを示す.

1.2.1　スマートフォン，タブレットにおける普及

スマートフォンでは，2008 年に HTC Dream / Google Dev Phone 1 （HTC）が発売されて以後，多数の Android 端末が発売された．世界的に見た場合，Droid （Motorola），Galaxy S （SAMSUNG）の販売拡大により，シェアを大きく伸ばした.

日本においては 2010 年より，Xperia SO-01B（ソニーエリクソン，NTT ドコモ），IS03（シャープ，KDDI）といった端末の発売により，Android の普及が進んだ．Android の普及は日本国内の携帯電話産業に影響を与えた．2010 年秋以後，国内の携帯電話キャリア会社から発売される新機種のうち，スマートフォンが占める割合が大幅に増加，その多くが Android を採用している．また，国内の携帯電話を製造するメーカにおいては，特殊な日本の携帯電話市場からの依存解消，より巨大な海外の携帯電話市場への再進出の手掛かりとしているところもある [6].

Android は高機能なスマートフォン向けの展開に限らず，新興市場向けの廉価な端末にも普及が進んでいる．後述するように，Android は構成するソフトウェア・コンポーネントの多くが無償で入手可能である．そのことから，廉価な性能を持つ端末に搭載し，安価に販売する例もある．代表例として IDEOS U8150（Huawei）が挙げられる．

このような背景から，スマートフォン市場において Android は急速にシェアを伸ばしている．先行して展開してきたスマートフォン向けプラットフォームには，Symbian，BlackBerry，Windows Mobile，iOS とあるが，これらを上回る伸びを示している．ある調査では，先行しスマートフォン向けプラットフォームにてシェアトップであった iPhone /iOS のシェアを上回ったとの結果 [7] がある．

1.2.2　Android の利用用途の拡大

Android は主に携帯電話での利用を想定したプラットフォームであったが，プラットフォームを流用したスマートテレビ Google TV[8] が 2010 年に登場した．スマートテレビとは，既存のテレビ，セットトップボックスに対し，インターネット接続機能やアプリケーション実行環境，メディア再生機能を付加するものである．

Android を搭載したタブレット・デバイスも発売された．そして Android3.0 [9] では，タブレット・デバイス向けに最適化したユーザ・インタフェースへ大幅な変更を行った．後続の Android4.0 [10] では，スマートフォン，タブレット間でのプラットフォーム統合が図られた．アプリケーションの実行基盤は同一であるため，既存の Android アプリケーションも利用できる．

組み込み機器向けの環境として，Open Embedded Software Foundation [11] による Embedded Master [12] が挙げられる．Embedded Master は，組み込み機器向けにカスタマイズを施した Android である．Android の API と互換を持たせつつ，デジタルテレビや DLNA，IP 電話といった情報家電向けの機能を拡張している．そして，Android を構成する各コンポーネントを細かく分割し，依存関係を維持しつつ，標準の Android に比べてコンパクトな構成に変更できる OESF Platform Builder を公開している．これは Android のシステムイメージのサイズを最大 60~ 65% 縮小した構成にすることができる [13][14].

これらの背景のもと，セットトップボックスや多機能フォトフレーム，デジタルサイネージと，Android はさまざまな目的に応用されている．

1.2.3　用途の拡大の背景

用途の拡大の背景には，Android を構成する多くのコンポーネントが Android Open Source Project (AOSP) としてオープンソースで公開されていることが挙げられる [15].

図 1.1 に Android のアーキテクチャ図を示す [16]．主に次に示すオープンソース ソフトウェアから構成されている．

Linux カーネル　オペレーティングシステム．

Bionic　BSD libc 由来の標準 C ライブラリ．Linux で広く用いられている glibc ではない．

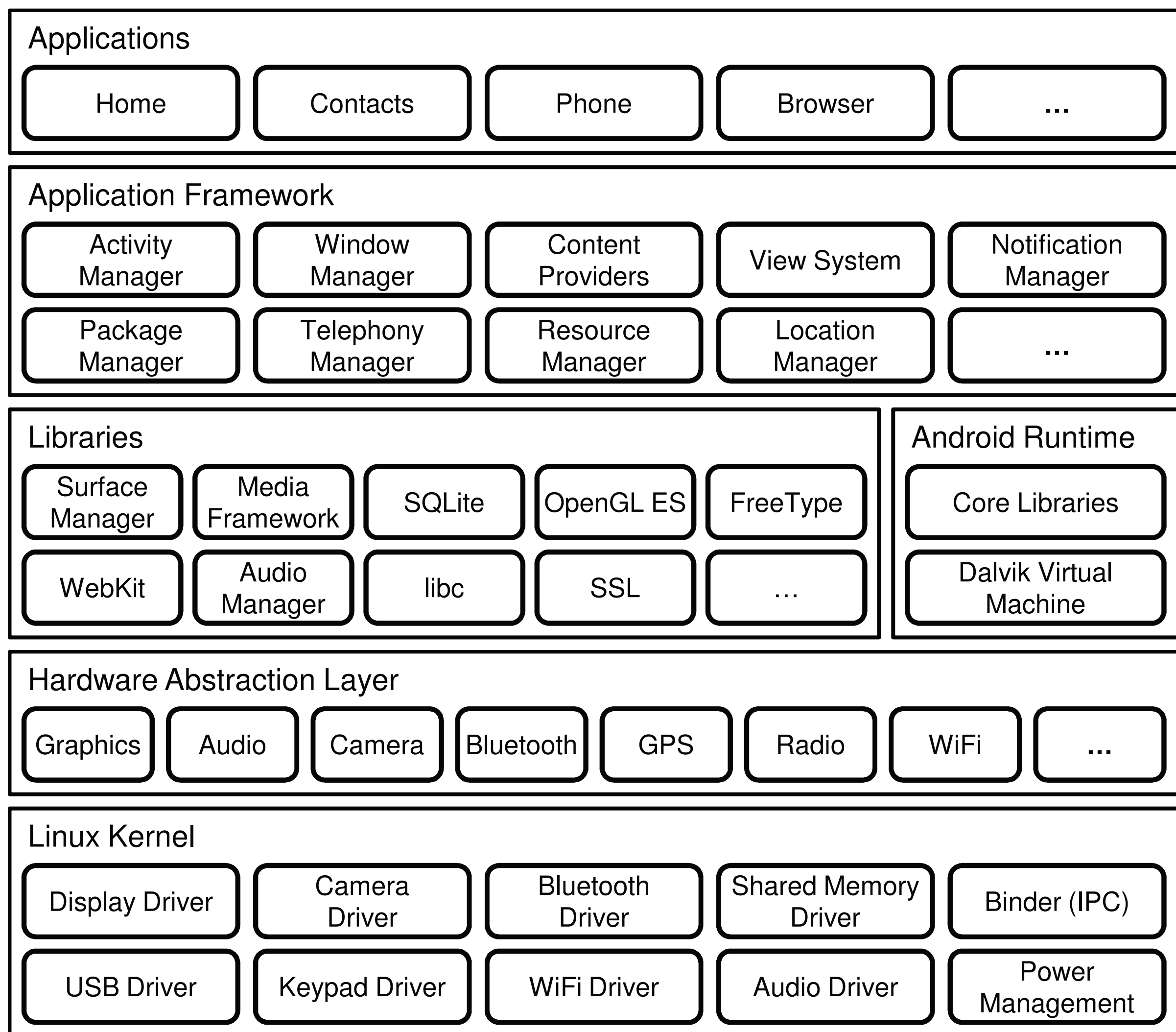

図 1.1: Android のアーキテクチャ図

OpenGL ES 組み込み機器向けのグラフィック API.

WebKit ウェブブラウザエンジン.

SQLite データストア基盤．軽量な RDBMS 実装.

Apache Harmony Java クラスライブラリ.

そして，Google が開発した部分である Android アプリケーションのランタイム，フレームワーク，標準アプリケーションの多くは， Apache ライセンスの下で配布されている[1].

[1]一方，Gmail や Google マップ，Google Play 等のアプリケーションは，Google のプロプライエタリなソフトウェアである．互換性を検証する Compatibility Test Suite (CTS) をクリアし，所定の要件を満たした端末に対してこれらのソフトウェアがライセンスされる.

これらのコンポーネントから構成される，Android 搭載デバイスを開発するメーカは各々のノウハウをソースコードのライセンスの制限の下，公開することなく開発できる利点がある．Android を構成するコンポーネントのうち，GNU Public Licence，GNU Lesser General Public License により公開が義務付けられているのは，主に Linux カーネルと WebKit である．また，GPL ライセンスのデバイスドライバとユーザスペースを分離し，ライセンス的に分離可能なフレームワーク Hardware Abstraction Layer も用意されている．

実際に各社から発売されている Android 搭載スマートフォンには，独自のソフトウェア，ハードウェアを盛り込むことで，差別化を図っている．とりわけ日本市場においては，前述の IS03 に代表される端末のように，既存のフューチャフォンの機能を盛り込んだ端末が登場し，携帯電話市場におけるスマートフォンへの移行を加速させた．機能としては，赤外線通信，ワンセグ，非接触型 IC カード通信など，これまでの日本の携帯電話に盛り込まれてきた機能が挙げられる．

AOSP にて公開されているソースコードがサポートするアーキテクチャは ARM，x86，MIPS である．また，公開されているソースコードを基に，SuperH[17]，PowerPC[18] と複数のプロセッサアーキテクチャへ移植が進められている．

1.3　Android のアプリケーション・アーキテクチャ

このように，多数のアーキテクチャ上で動作する Android のアプリケーションは，ポータビリティを高めるために，Java 言語で記述される [19]．Java では，すべてのプログラムは一旦，中間言語である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，仮想機械（VM : Virtual Machine）上でバイトコードを逐次解釈し実行する．

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 Java バイトコードではなく，Dalvik バイトコード [20] と呼ばれる，独自の命令セットを持つバイトコードを採用する．両者は，前者がスタック・マシンの VM を想定した命令セットであるのに対し，後者はレジスタ・マシンの VM を想定した命令セットとなっており，まったく異なる．バイトコードの実行は，通常の Java VM ではなく Dalvik VM[20] と呼ばれる独自の VM 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・コードの実行に比べて遅い．そのため，Java バイトコードに対しては，これまでその実行を高速化する手法が研究，提案されてきた．そして Dalvik バイトコードにおいても，動作速度の低下を回避するために，いくつかの手法を盛り込んできた．

1.3.1　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 [21] [22]（Just-In-Time コンパイル．以下 JIT とする）や事前コンパイル [23]（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 では 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 9.2 で述べるが，コードがコンパイル前に比べて膨らんでしまうという欠点がある．バイトコードを構成する，仮想機械の命令セットは，プロセッサの命令セットに比べて抽象度が高い．1 つのバイトコードから，VM のインタプリタは，複数のプロセッサの命令列を用いてその動作を行う．そのため，バイトコードから同等の動作を行うネイティブ・コードへ展開する場合，バイトコードに比べてその大きさは増加する．これは汎用 PC に比べ大容量のメモリを搭載できない組み込みプロセッサにおいて，大いに問題である．

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 による翻訳を介しないことで，オーバヘッドを減らすことができる．さらにバイトコードを直接実行することから，JIT や AOT のようにネイティブ・コードを生成せず，メモリ，ストレージを圧迫しない．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．

1.3.2 Dalvik バイトコードに対する高速化手法の現状

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．Android2.2 において Dalvik VM へ JIT が組み込まれたほか，サードパーティによる JIT [24] 程度である．

現在 Android において主流となっている高速化の手法は，ネイティブ・コードを呼び出す方法である．Dalvik VM には，Java VM 同様に Java Native Interface（JNI）を介した，ネイティブ・コードの呼出方法を持つ．Android のアプリケーションは，ネイティブ・コードで記述された Android のコアライブラリへのアクセスに JNI を用いている．同様に，アプリケーションは NDK（Native code Development Kit）を利用し独自のネイティブ・コードを用意すれば，JNI を通じて呼び出すことができる．この手法は，高い動作速度，低い遅延，効率を要求するゲームやメディア再生，ウェブブラウザといったアプリケーションで用いられている．

しかし，アプリケーションにプロセッサ依存のネイティブコードを組み込むため，VM が隠蔽していたハードウェアの差異が表面化し，アプリケーションの汎用性は低下する．現在，多くの Android デバイスが稼働するアーキテクチャは ARM であるが，Android 自身が公式に x86，MIPS アーキテクチャ向けにもメインテナンスが行われ，用途の拡大とともに，稼働アーキテクチャが ARM とは限らなくなっている．実際に CTS を通過し，Google Play 等を利用可能な MIPS アーキテクチャの Android デバイスが登場した．

これに対し Google Play では，アプリケーション配布元が，稼働アーキテクチャ，ディスプレイサイズ，Android 自身のバージョンに応じてダウンロードするパッケージファイルを切り替える仕組みを提供している [25]．しかし異なるデバイスモデルに応じたパッケージの配布は，アプリケーションのコンパイルから検証，配布まで，広い過程において工数の増加を意味する．よって，根本的な解決とは至っていない．

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，バイトコードのハードウェア・アクセラ

レーションが有効な手段と考えられるが，しかし，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 [26]．しかし，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 [27]．

1.4 研究の目的

現在の Dalvik VM の動作性能の低さ，これまで Java において提案されてきた高速化手法の背景を踏まえ，Dalvik VM へのハードウェア・アクセラレーションを適用する．

その過程で，本体プロセッサやメインメモリの増加，プログラムの変更量を調査し，その妥当性を検証する．併せて，従来のハードウェアによる Java バイトコードの高速化同様に性能が向上するか検証する，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．

1.4.1 Dalvik デコーダの実装

Dalvik バイトコードのハードウェア・アクセラレーション方式には，Java VM において有力な Jazelle 方式を採用する．追加するハードウェアは，Dalvik バイトコード解釈し，ネイティブ・コードを出力する専用デコーダのみに留める．Dalvik バイトコードより変換したネイティブ・コードのデコード，実行は，プロセッサ内の既存の資源を利用する．

1.4.2 DRMT 機構の効果の測定

Dalvik VM のインタプリタならびに Dalvik バイトコードから生成されたネイティブ・コードは，Dalvik VM における演算対象となる Dalvik レジスタと呼ばれる領域と物理レジスタとの間で，バイトコードのデコード毎にロード/ストアを行っている．連続するバイトコードにおいて，同一の Dalvik レジスタが演算対象になっている場合においても，演算結果のストア，直後にロードといったように，無駄なメモリアクセスが生じている．

バイトコード毎に Dalvik レジスタを物理レジスタと主記憶間でロード/ストアするのではなく，最近物理レジスタにロードした Dalvik レジスタについてバイトコード間をを横断して複数保持する．これにより，すでに物理レジスタに配置されている Dalvik レジスタについてロード/ストアを省く．それにより，よりハードウェア・アクセラレーションを高速にする．

1.4.3 遷移コストの削減

Dalvik アクセラレータはすべての Dalvik バイトコード単体で直接実行することは困難であることから，プロセッサ・ネイティブな命令の実行モードとの切替，そして Dalvik VM との間で制御を移すことは少な

くない．この Dalvik アクセラレータと プロセッサ・ネイティブな下で動く Dalvik VM との間で切り替わる，実行モードの遷移に必要なプロセッサのサイクル数は Dalvik アクセラレータの性能向上を阻害する．この影響の大きさを示した上で 2 つの削減手法を示す.

一つは，Dalvik アクセラレータではデコードできない命令を Dalvik VM に制御を移して実行する際，次のバイトコードの種別に問わず Dalvik アクセラレータへ制御を戻す動作を変更し，Dalvik VM 側でもアクセラレーションの可否を判断させる．もう一つは，2 つの実行モードにて役割の異なる物理レジスタについて 2 重に設ける．モード遷移に応じてもう一方に高速スイッチすることで，従来個々にロード/ストアしていた動作を削減する．

1.5　本研究が与える影響

これまで挙げたように，現在の Android の普及状況，多様な利用動向が見られる．アプリケーションにおいては， Dalvik VM によりデバイスの差異を吸収でき，多彩な環境で動作させることができる．一方で性能の面に Dalvik VM がオーバヘッドになっている．そしてこの問題への対処手段である，NDK がデバイスの分断や検証の増加を招いている．

このような中，Dalvik アクセラレータを実現することによって，低クロック，低消費電力で動作する Android デバイスを実現できるとみられる．現在のスマートフォン向け半導体は，性能の向上が著しい一方で，Android 搭載端末では Huawei 社 IDEOS のように性能を抑えて安価に販売する製品も現れている．このような安価な端末の販売，普及において Dalvik アクセラレータは有用とみられる．

また，小規模な組み込み機器向けへの応用も考えられる．ソフトウェアについては Embedded Master のように必要なメモリ，ストレージを減らすカスタマイズ手法と，Dalvik アクセラレータを併用することで，性能の限られた組み込みシステムにおいても，Android アプリケーションをより高効率に実行できるものとみられる．

1.6　論文の構成

本論文の構成は以下のようになっている．

第 2 章では，本論文の前提知識として，まず Java VM の内部構造について述べる．Java アプリケーションの実行方法から，Java VM 上での動作を示す.

第 3 章では，Jazelle DBX をはじめとする，既存の Java における高速化手法について説明する． ソフトウェアによる手法から，複数種のハードウエアによる手法について取り上げる．

第 4 章では，Dalvik VM のアーキテクチャについて述べる．Android における Dalvik VM の役割，VM の内部構造を述べたのち，Java VM と異なる点について示す.

第 5 章では，本論文で提案する Dalvik アクセラレータについて述べる．まずアクセラレータの構造を示す．そして，バイトコードよりネイティブ・コードに変換，プロセッサパイプラインへ出力する動作について説明する [28].

第 6 章では，Dalvik アクセラレータが発行するネイティブ・コードより，Dalvik レジスタのロード/ストアを行う，メモリ・アクセス命令を削減する機構，DRMT について説明する．そして DRMT の動作例を挙げ，Dalvik アクセラレータが発行しようとする命令列より，Dalvik レジスタへのメモリ・アクセス命令を削減する手法を示す [28].

第 7 章では，Dalvik アクセラレータを単純に実装した際，実行モード遷移時のコストが Dalvik アクセラレータの性能に及ぼす問題を示す．それに対し，遷移の回数を削減する方法，遷移時の物理レジスタの保存復帰処理の削減による，遷移コストの削減手法について示す [29].

第 8 章では，提案する Dalvik アクセラレータを組み込み，評価を行う環境として用いる SimMips について示す．実装先として SimMips を用いた背景，SimMips に Dalvik アクセラレータを組み込み Android を動作させるために必要な手順，SimMips に対して行った変更を示す．そして現在の携帯情報端末向けプロセッサに近い，妥当な性能が発揮されるか確認する [30].

第 9 章では，今回提案する Dalvik アクセラレータのうち，DRMT によるメモリ・アクセスの削減の評価を行う．バイトコードより単純にネイティブ・コードを出力する場合，Dalvik VM に搭載されている JIT を用いた場合に比べて，Dalvik アクセラレータが効率的な命令を出力していることを示す [28].

第 10 章では，本論文で示した結果をまとめ，今後の研究の方向性について示す.

第2章　Java Virtual Machine

本章では，Dalvik VM の説明ならびに Dalvik アクセラレータの実装について説明することを踏まえ，前提となる Java VM[31] のアーキテクチャについて説明する．続いて次章では，Java 向けの高速化手法について説明する．

2.1　Java VM による Java プログラムの実行

Java VM は Java プログラムを実行する仮想機械である．Java 言語で記述されたプログラムから，Java コンパイラにより**クラス・ファイル**と呼ばれる，Java バイトコードを含むバイナリが生成される．そして Java VM はインタプリタで Java バイトコードを翻訳し，動作するハードウェアやオペレーティングシステムに合わせた命令へと逐次変換しアプリケーションを実行する．よって Java プログラムは Java VM により，ハードウェアやオペレーティングシステムの差異を吸収できる．

Java VM は Java の設計元である Sun Microsystems（現 Oracle）による標準の実装をはじめ，HotSpot，Apache Harmony，Kaffe と複数の実装が存在している．そして，2007 年には JDK が GPL ライセンスの下でオープンソースへ移行し，OpenJDK として展開している．

2.2　Java バイトコード

Java VM は物理的には存在しない計算機プログラムであるが．しかし，一般的なプロセッサにおける命令セットの定義と同様に，命令セットを持つ．Java バイトコードは Java VM の命令セットと見ることもできる．Java バイトコードでは 200 個の命令が定義されている（参照が解決済のオブジェクトを高速で操作できる，サフィックスに`-quick` が含まれたバイトコードも含めると，227 命令となる）．

Java バイトコードは，スタックの push/pop により演算，制御を行う，スタック・マシンを想定した命令セットである．演算やその結果の書き込み対象はスタックの上部に限定され，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，Java バイトコードの 1 バイト目の内容はオペコードと共通であるが，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[1]．このことから，オペランドの指定がなく，オペコードのみ 1 バイトで構成される命令がある．オペランドを指示する命令が少ないことから，命令当たりのバイトコードの長さはレジスタ・マシンに比べて短くなる傾向がある．

現在のプロセッサはその多くがレジスタ・マシンであるため，スタックをどのように扱うかは VM の設計による．単純にメモリ上にスタックを置くのではメモリアクセスが頻繁に発生してしまうため，多くの

[1] C 言語における switch – case 構文に相当する，lookupswitch，tableswitch 命令は分岐先リストが続く限り無限長である．

VM では，頻繁にアクセスされるスタックの上位数個を物理レジスタ上に配置する設計を用いることがある．これによりスタックのアクセスに伴う，ロード・ストアを軽減する設計を施している．

2.3 Java バイトコード・インタプリタ

インタプリタが Java バイトコードを実行する様子を図 2.1 に示す [32, 33]．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．

2.3.1 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度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，ここに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．

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]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．

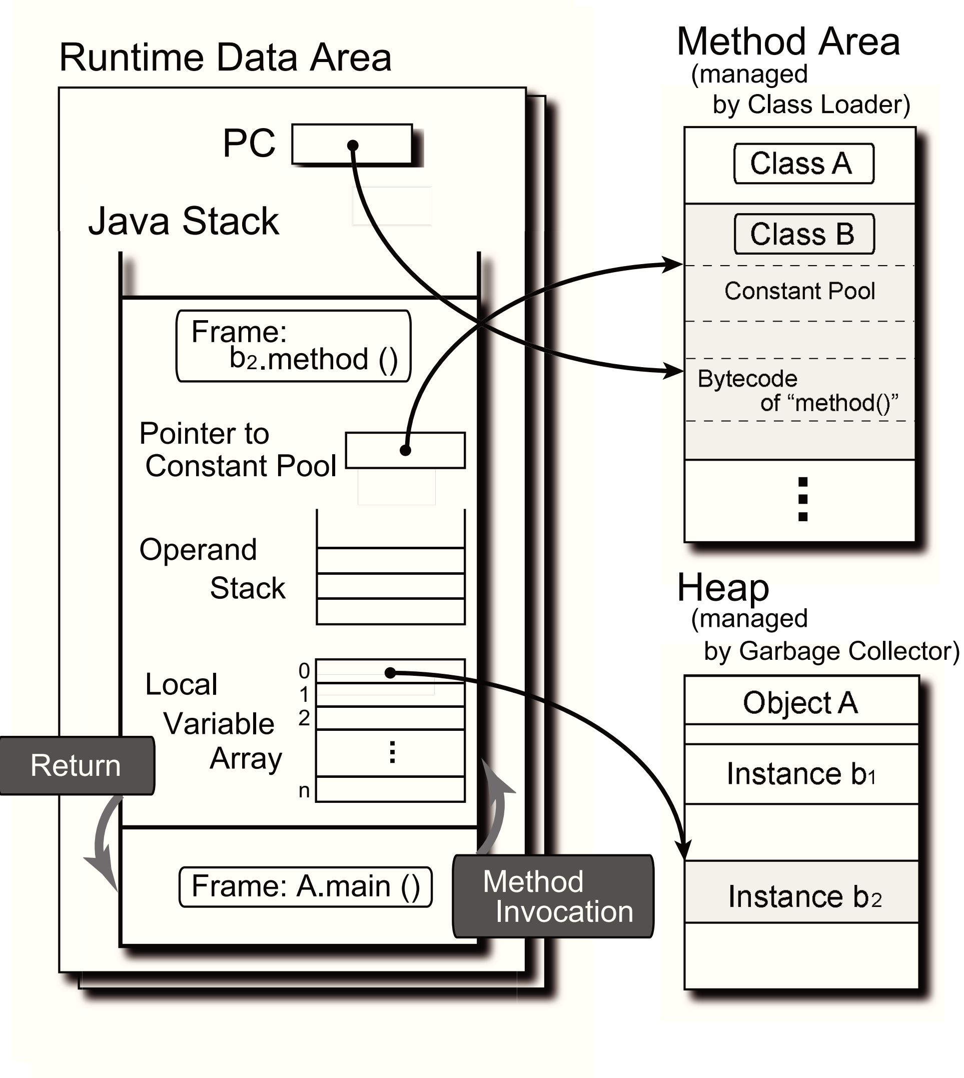

図 2.1: インタプリタによる Java バイトコード実行

2.3.2　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 2.1 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．

2.3.3　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 はスレッド毎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度にフレームを push し，復帰する度に 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 番は常にオブジェクトの参照に対応する．

2.4 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 2.1 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，オペコードにしたがって演算を施す．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．

2.5 まとめ

本章では，Java を構成する Java VM，Java バイトコードについてその概要を示した．Java VM は Java バイトコードを実行する仮想機械であり，逐次翻訳することで Java プログラムの実行を行う．Java バイトコードは一般的なプロセッサの命令セットと比べて，抽象度が高い複雑な命令を有している．

第3章　関連研究: Javaにおける高速化手法

Java VM は，Java バイトコードの実行，GC をはじめとするメモリ管理やベリフィケーションなど，様々な処理を行っている．各々の実行工程について Java VM においては，これまで様々な高速化手法が提案されてきた．本章では，Java バイトコードの実行速度を高速化する手法について既存の例を示す.

3.1　ソフトウェアによる方式

ソフトウェアによるバイトコードの高速化手法は古くから考案されている．その多くは VM によるバイトコードの翻訳に替わり，バイトコードから実行するプロセッサに対応する機械語を生成し，それを実行するという手法である．ネイティブコードの生成タイミングに応じて，大きく分けて JIT (Just in Time) コンパイルと AOT (Ahead of Time) コンパイルに分類される.

3.1.1　JIT コンパイル

JIT コンパイルはプログラムの実行時にバイトコードからネイティブコードを生成する手法である．よって JIT を担当するスレッドは，通常 VM のインタプリタスレッドと並列に動作する．並列に動作することから，動的な VM の動作状況を取得することができる．取得した動作状況より，頻繁に実行される個所に絞ってネイティブコードの生成を行える.

JIT はネイティブ・コードを生成する単位・範囲により，2 種類の手法がある．メソッドベース JIT は，メソッド単位でネイティブコードを生成する．トレースベース JIT は頻繁に実行されるバイトコードの列を一つの「トレース」として扱い，トレース単位で JIT を行う.

トレースベース JIT を用いた Java VM の例として，HotSpot[21]（Sun Microsystems）が挙げられる．HotSpot は，プログラム中の頻繁に実行される「ホット・スポット」を検出し，JIT コンパイルを行う．実行中の VM から得たプログラムの実行頻度に応じて，最適化の強度を変えている．これにより，JIT によるネイティブ・コードを生成するオーバヘッドを抑えることができ，VM のスレッドへの影響を軽減する.

欠点として，ネイティブコードの生成により，実行中の VM から CPU リソースを奪う点である．JIT がネイティブコードを生成する時間が，生成したネイティブコードの実行に切り替えたことによる時間削減より上回ってはならない．よって，頻繁に実行される個所に対し優先的に JIT を行うことが望ましいとされる.

3.1.2 AOT コンパイル

AOT コンパイルは，プログラムの実行前にバイトコードからネイティブコードを生成する手法である．JIT コンパイルは，少なからず実行中の CPU 時間を消耗するが，AOT コンパイルでは，プログラムの実行中に CPU 時間を消耗することはない．プログラムの実行前に十分な時間があれば，プログラム全体をネイティブコードにできる，また JIT に比べより高い最適化を行える．

Java での実装例としては，GNU Comipler for Java（GCJ）が挙げられる．通常の Java コンパイラは Java ソースコードより Java バイトコードを含む Java クラスファイルを生成するが，GCJ は直接実行ターゲットの機械語を生成することができる．オブジェクトファイルの出力も可能で，ほかの言語からコンパイルされたオブジェクトファイルとの間で呼び出し，リンクすることも可能である．

このほかに AOT が用いられている手法としては，.NET Framework のネイティブ・イメージ・ジェネレータ（ngen.exe） が挙げられる．こちらは .NET Framework のインストール，アップデート時に，中間言語 CLI（Common Language Interface） で記述された主要なライブラリを ngen が AOT コンパイルする．

3.1.3 命令セットによるネイティブコード生成支援

異なるアプローチでは，Jazelle RCT[34] で用いられる Thumb-2EE と呼ばれる命令セットがある．ARM アーキテクチャには複数の命令セットを切り替え可能なものがあり，Thumb-2EE はこの一つのモードとして動作する．

Thumb-2EE の命令セットは，バイトコードにおいてよく用いられる操作を命令セットに盛り込んでいる．その例として，`CHKA` 命令が挙げられる．これは，配列の添え字番号が溢れていないかチェックし，例外を発生させる．Java における ArrayIndexOutOfBoundsException 例外のような，配列の大きさよりも大きい添え字番号へのアクセスに対する例外に用いられる． 通常に JIT コードを生成した場合，比較を行い，その結果に応じて例外を発生させる命令列を出力するのに対し，コード長を抑えている．

このように， Thumb-2EE 命令セットで JIT，AOT コードを生成することで，出力されるネイティブコード長を抑える，命令の実行速度の向上を行っている [35].

3.2 ソフトウェアによるネイティブコード生成の課題

これまで取り上げてきた，ソフトウェアによるバイトコードからネイティブコードを生成する手法には，生成したネイティブコードをどこに配置するかという問題がある．JIT の場合は実行中であるため主記憶へ，AOT の場合は事前に生成するので補助記憶へ，それぞれネイティブコードを保持する必要がある．よって，より多くメモリ，ストレージを使用する．

バイトコードはプロセッサの命令セットに比べて，抽象的で複雑な動作を行う命令が存在する．それらは VM で翻訳され，複数のネイティブ・コードの命令列を実行することで実現している．

従って JIT，AOT 何れにおいても，VM が行ってきた動作と同種の振る舞いを行うネイティブコードを生成しようとするのならば，元のバイトコードを上回る大きさのネイティブコードが生成されることとなる．

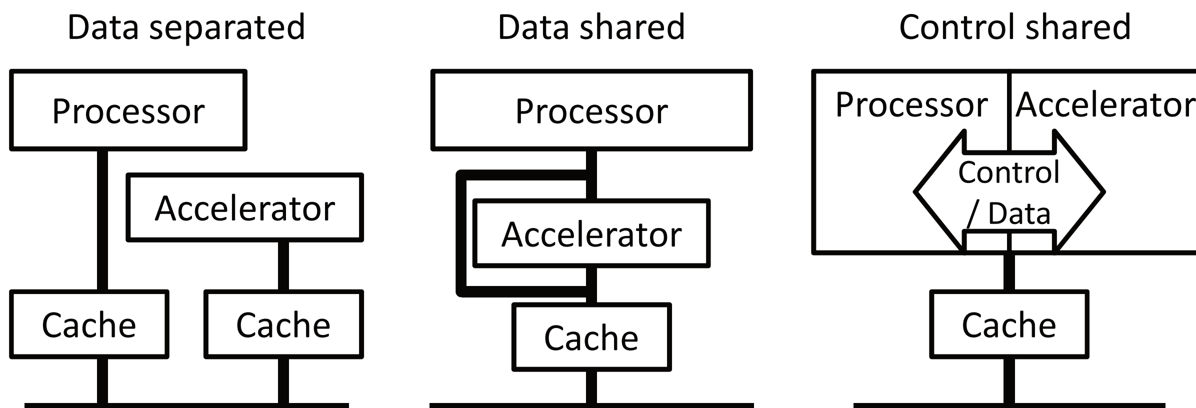

図 3.1: Java アクセラレータの実装方式

3.3 ハードウェアによる方式

次に，バイトコードをハードウェアによって解釈することで，高速化を行う手法について挙げる．本論文ではこの手法を **Java アクセラレータ**と呼ぶ．Java アクセラレータは VM 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]．

Java アクセラレータの実装方法は，いくつかある．水野らが分類に用いた 3 つの実装方法 [37] を基に，ハードウェアによる方式を挙げ，各々の特徴を示す．以下の実装方式と，その実装例より順に示す．その最後で，本論文で実装する Dalvik アクセラレータの実装の基となる，Jazelle 方式 [38, 39, 40, 41, 42, 43] について詳しく述べる．

図 3.1 に，各実装方式におけるプロセッサとアクセラレータの接続関係を示す．

データ分離型 コプロセッサのように，主となるプロセッサとは独立したプロセッサ機能を有する．コプロセッサバスやメモリマップド I/O を介して，アクセラレータを制御する．

データ共有型 データバスを共有するする方式．データバスに介在し，バイトコードをネイティブ・コードに変換してプロセッサへ入力する．接続方式の制約上，バイトコードを読み出してからネイティブ・コードを CPU へ出力するまでに数サイクルかかる．

制御共有型 アクセラレータはプロセッサに組み込まれ，プロセッサの制御も行う．単純に変換した命令列を送り出すだけでなく，プロセッサの動作を読み出し，その内容からより緻密なアクセラレーションを可能とする．

3.3.1 JVXtreme

JVXtreme は inSilicon 社による Java アクセラレータである．実装形態はデータ分離型となる．ホスト CPU と独立したプロセッサであり，Java バイトコードを実行可能な Java プロセッサと見ることができる．

[1]GC もハードウェアで行うアクセラレータも一部ある [36].

ホスト CPU との接続は，コプロセッサ・インタフェースか，メモリバスを共有しメモリ・マップを用いる．

ホスト CPU とは独立したプロセッサではあるが，JVXtreme が直接実行できる Java バイトコードは 92 命令である．これは後述の他のアクセラレータに比べて少ない．単体ですべての Java バイトコードが実行できないため，実行できないバイトコードについて JVXtreme は動作を停止させ，ホスト CPU に制御を移す．そしてインタプリタで実行を継続する．インタプリタでの実行が終わったら，再び JVXtreme へ制御を移す.

3.3.2 picoJava

picoJava[44][45] は Sun Microsystems による Java のプロセッサ実装である．実装形態はデータ分離型となる．アクセラレータではなく独立したプロセッサであり，Java バイトコードを単体で実行することができる．1999 年には後続のプロセッサ picoJava-II も発表されている．

プロセッサ内部は picoJava-I で 4 ステージ（フェッチ – デコード – 実行・キャッシュ – ライトバック），picoJava-II で 6 ステージ（フェッチ – デコード – レジスタアクセス – 実行 – データキャッシュアクセス – ライトバック）のパイプライン構造からなる．一般的なプロセッサのレジスタに替わり，64 エントリのスタック・キャッシュを持つ [46][47].

picoJava の特徴として，命令畳み込み（Instraction Folding）が実装されている点がある．これは Java バイトコードから特定の命令の組み合わせを認識し，冗長なオペランドへの操作を削除するものである [48]. 通常 Java バイトコードにおいて演算を行うとき，その動作を行うバイトコードは，値をオペランド・スタックにプッシュ，演算，その結果をポップ，と 3 命令からなる．この動作に命令畳み込みを適用すると，オペランド・スタックへのプッシュと演算を一括して行うようになり，命令の実行数を短縮する．

図 3.2 に命令畳み込みの例 [44] を示す．ここでは，ローカル変数 0 番からオペランド・スタックへプッシュする `iload_0`，スタックの上位 2 つの間で整数加算演算を行い，その結果をオペランド・スタックへプッシュする `iadd` の組み合わせで示す．命令畳み込みがない場合，Cycle 1 にてローカル変数をプッシュしたのち，Cycle 2 にて加算を行っている．

一方，命令畳み込みを用いるとこの一連の動作を検知し，オペランド・スタックのトップとローカル変数 0 番との間で直接演算を行う．そして加算結果をオペランド・スタックのトップに書き込む．一連の動作は Cycle 1 で完結する．このように命令畳み込みはオペランド・スタックへの操作と演算の組み合わせに対し，冗長なオペランド・スタックの操作を排除し動作を高速化する．

家電や組み込み機器への利用を想定しており [46]，picoJava-II を搭載した富士通のプロセッサ MB92901 では，内部には割り込みや電源管理を備えている．

picoJava-II はのちにオープンソースとして設計が公開され，FPGA への実装例が存在する [49].

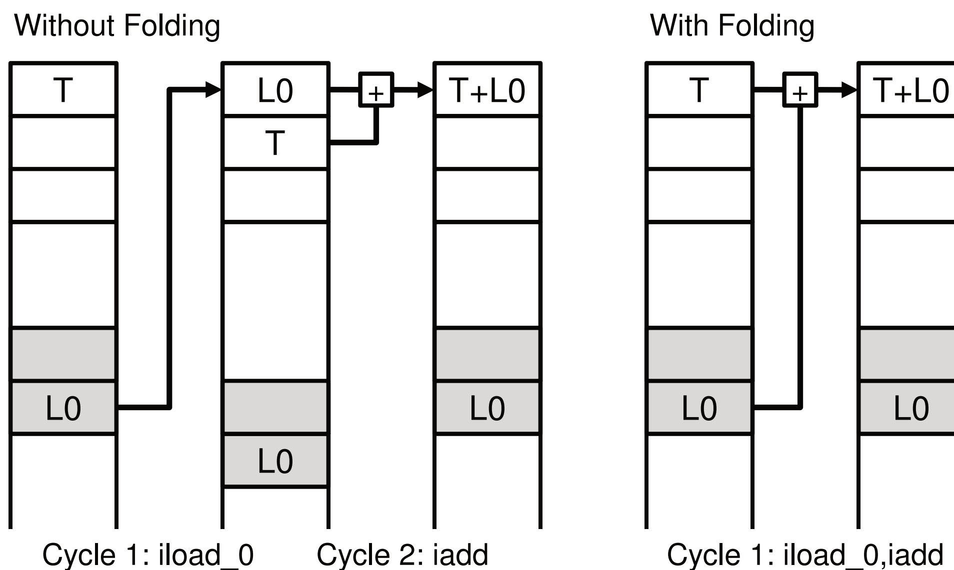

図 3.2: 命令畳み込みの例

3.3.3 JSTAR

JSTAR[50] は，Nazomi Communications による Java アクセラレータである．実装形態はデータ共有型となる．図 3.3 に JSTAR のブロック図 [41] を示す．

JSTAR はデータバス入力よりバイトコードを入力する．そして変換してデータバス出力にネイティブ・コード出力し，CPU に入力する．バイトコードの読み出しは CPU からアドレスバスの値を取得し，CPU のプログラムカウンタと JSTAR 内部にある Java プログラムカウンタの値を多重化して，アクセスする先を決定する．入力されたバイトコードからネイティブ・コードを生成するため，JSTAR 内部の 2 段パイプラインにより，ネイティブ・コードの生成は 2 サイクル遅れる．

JSTAR は Java バイトコードを 160 命令サポートし，ハードウェアで処理する．それ以外の命令については，CPU に制御を移して インタプリタで実行を継続する．

JSTAR を接続する箇所にもよるが，キャッシュを CPU と共有する形式となる．キャッシュがアクセラレータよりもメモリバス側にある場合，変換前のバイトコードがキャッシュされることになる．展開後のネイティブ・コードよりもキャッシュを占有する領域を抑えられる．よって，より多くのバイトコードをキャッシュすることが可能である．

3.3.4 BTU

BTU(Bytecode Translation Unit)[37] は，ルネサステクノロジ（現 ルネサスエレクトロニクス）による，SH-3 Mobile，SH-4 向けの Java アクセラレータである．実装形態は制御共有型となる．BTU のブロック図を図 3.4 に示す．BTU は SH-3 Mobile，SH-4 プロセッサ内部に組み込まれる形式で実装される．

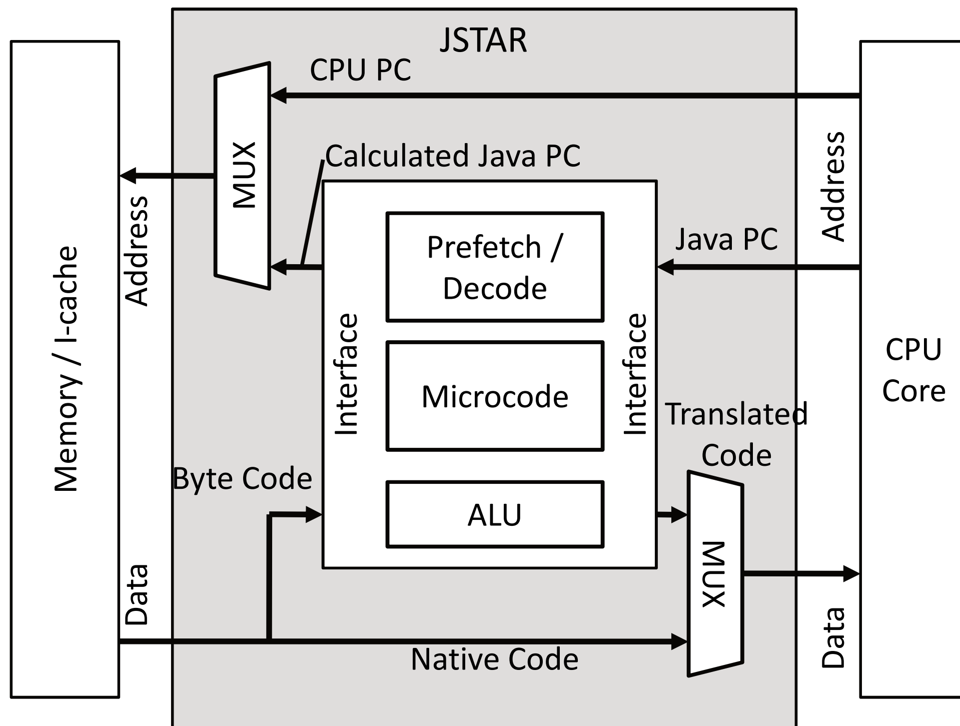

図 3.3: JSTAR のブロック図

これまで挙げてきた Java アクセラレータとは異なり，プロセッサの制御と接続されているのが制御共有型の特徴である．CPU 側の動作を検知し，それに応じたより細かい制御ができるのが特徴である．

BTU は動作モード切り替えて無条件分岐する命令を有し，実行されると BTU は活性化する．BTU が有効になると，CPU の命令バッファから送られた バイトコードをフェッチして（Byte-code Fetch），バイトコードをデコードする（Translation Logic）．Translation Logic からは，バイトコードの動作に対応する，16 ビット長の SH 命令セット (native code)，拡張制御コード (extended code)，即値 (immidiate) の制御信号のセットを出力する．

BTU と CPU 間を繋ぐ BTU-bus は例外制御とレジスタファイル間で接続されている．このバスを通じて，配列の境界チェックなどの例外の検出を CPU でなく BTU が行う．これにより CPU が例外の検出を行わなくて済み，高速な実行に貢献する [51].

BTU は Java バイトコードを 159 命令サポートし，ハードウェアで処理する．それ以外の命令については，実行モードを戻し，CPU に制御を移して インタプリタで実行を継続する．

3.3.5 Jazelle DBX

Jazelle DBX は ARM による，ARM アーキテクチャ向けの Java アクセラレータである．実装形態は制御共有型となる．本論文で言及する Dalvik アクセラレータの実装手法は Jazelle DBX を基とする．Jazelle DBX

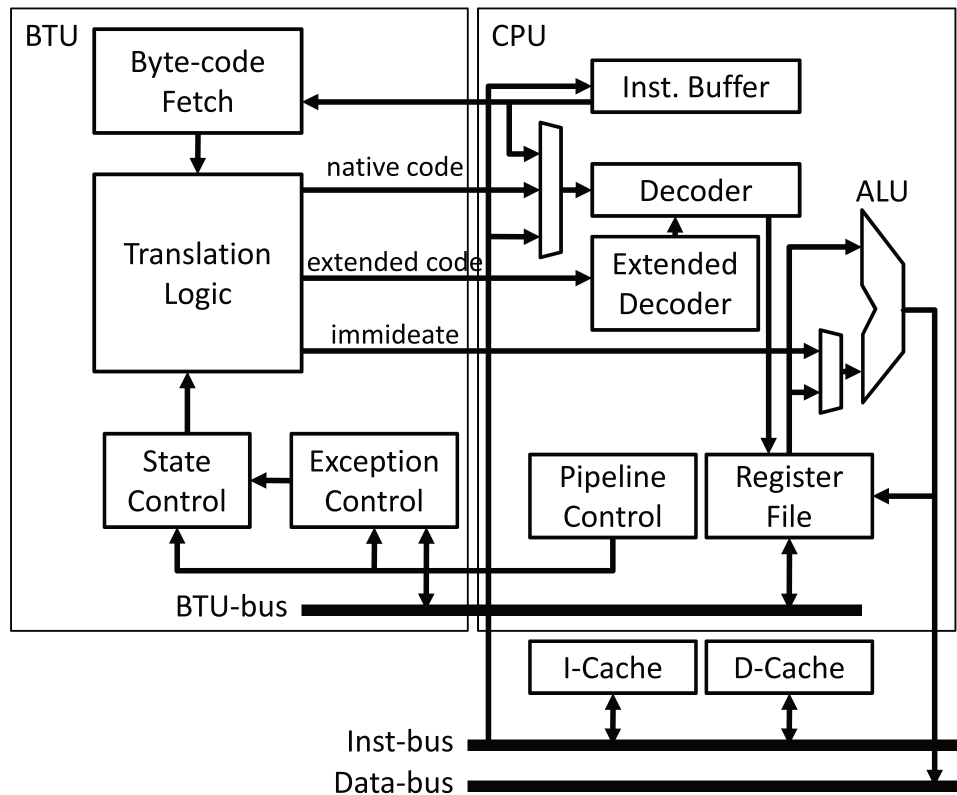

図 3.4: BTU のブロック図と CPU との接続関係

はプロセッサのパイプライン上にバイトコードからネイティブ・コードを生成するデコーダが組み込まれる．便宜上このアクセラレータ方式を，今後 **Jazelle 方式**と示す．

Jazelle 方式を適用した場合のプロセッサ構成を図 3.5 に示す[2]．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[3]．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する．そして 2 つのデコーダはステータス・レジスタにより選択され，プロセッサが解釈する命令セットを切り替える．

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 バイトコードのある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 バイトコードのみ解釈する．ARM モードに戻る命令は

[2]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 [38, 39]．図は，Paul らの実装 [43] を参考にした．

[3]ARM アーキテクチャには，命令幅を縮小した Thumb 命令セットに対応したモードがあるが，ここでは省略する．

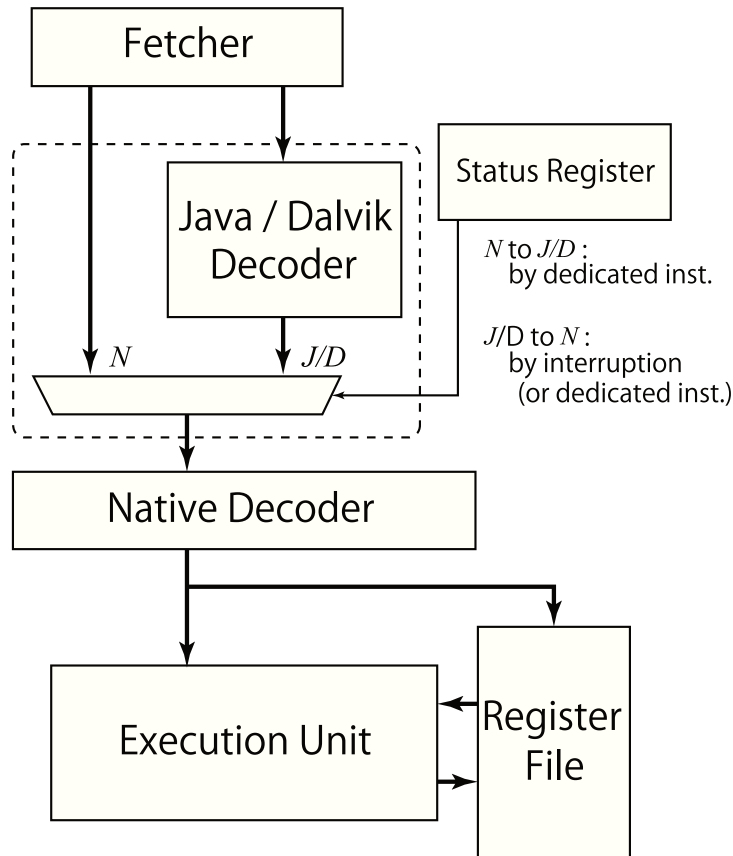

図 3.5: Jazelle 方式のブロック図

ない．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 3.1 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．よって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（*r*0 ~ *r*3）．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， 図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．

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，通常通り 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 [42].

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 [39] としている．

3.4 Java アクセラレータの比較

今回挙げた Java アクセラレータは，いずれも直接実行できる Java バイトコードの命令数は異なる．表 3.2 に主なアクセラレータが扱える，Java バイトコードの命令数，組み込みデバイス向け Java ベンチマーク

表 3.1: Jazelle DBX におけるレジスタ割り当て（一部）

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

Embedded CaffeineMark[52] のスコア[4]，おおよその回路規模となるゲート数を示す [42][53]．表中の Java バイトコードの命令数は，`-quick` 命令も含めた，227 命令が最大となる．

一般に，独立した実装形態に近いほど，クロック当たりの ECM スコアは向上する傾向にある．表中で ECM スコアが高い JVXtreme や picoJava-II はいずれもコプロセッサ型である．JVXtreme を除き，アクセラレータで扱えるバイトコードの数は，ECM スコアに比例する傾向がみられる．

一方で，実装命令数，すなわちアクセラレータで実行できるバイトコードが多ければ，高速で実行できるとは限らない．ECM スコアと実装命令数の間には，相関がみられない．実装命令数との相関が強いのは，ゲート数である．

Jazelle DBX を基準に比較すると，ほかのアクセラレータに比べて，使用するゲート数が著しく低い．制御共有型はプロセッサに組み込まれるため，できるだけ小型であることが望ましい．その面では，Jazelle DBX は同じく制御共有型である BTU に比べて高効率である．

3.5 まとめ

本章では Java の高速化手法として，ソフトウェア，ハードウエア双方の方式での事例を挙げた．そしてソフトウェアによる高速化の問題点として，バイトコードに比べて膨張する，ネイティブ・コードの保持先の増加について示した．その問題を踏まえ，ネイティブ・コードの膨張がなく，バイトコードを直接実行可能なハードウェア方式について，種類の異なる Java アクセラレータの実装例を示した．

表 3.2: 主な Java アクセラレータの比較

アクセラレータ	接続方式	ECM スコア [score/MHz]	ゲート数 [K]	実装命令数
JVXtreme	コプロセッサ型	9.10	35	92
picoJava-II	コプロセッサ型	9.30	120	227
JSTAR	データ共有型	6.00	27	160
BTU	制御共有型	6.55	75	159
Jazelle DBX	制御共有型	5.50	12	144

[4] スコアが高いほど高性能．

第4章　Dalvik アーキテクチャ

本章では，Dalvik VM について，Android 上における役割，バイトコードの演算対象であるレジスタ，バイトコードの仕様，VM のメモリ管理を説明する．そして，Dalvik VM はレジスタ・マシンの VM[54] 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．この Dalvik VM における相違点について示す．Dalvik VM は Android におけるユーザ・アプリケーションの実行基盤である．基本的に Android において，ユーザが操作するアプリケーションは，Dalvik VM 上で動作する．

4.1　Dalvik VM 採用の背景

Android においてアプリケーションの実行基盤に，既存の Java VM ではなく独自の VM である Dalvik VM を採用する背景は推測も含めていくつか存在する．そのうちの一つに，Google や 端末を販売するベンダが，Java ならびにその周辺ライブラリの利用に伴うライセンスの取得を回避するため，というのがある．

Dalvik VM が供給するクラス・ライブラリは Java 純正のものでなく，Apache Harmony のクラス・ライブラリを利用している．そこに Android の機能へアクセスするクラス・ライブラリを盛り込んでいる．

Java にてよく使われる基本的なクラス・ライブラリは揃っているものの，そのすべては含まれておらず，Swing や AWT といった GUI 向けのウィジェット・ツールキットも省かれている．標準の Java 構成である Java SE といったエディション や Java の携帯機向けセット Java ME (J2ME) の各プロファイルとも，互換性がない．

また，Sun の有する Java の知的資産を侵害しないよう．Dalvik VM はクリーンルーム方式での開発を行った [55]．その結果，Google や 端末を販売するベンダが，Java ならびにその周辺ライブラリの利用に伴うライセンスの取得の必要性を回避している．一方で，独自のバイトコード，VM，クラス・ライブラリを用いる Android の手法に対して Sun は 「Java を分断する」と批判している [56].

4.2　Dalvik バイトコードの生成

Android アプリケーションは通常，Java 言語で記述する．しかし，VM が解釈するバイナリは Java バイトコードではなく Dalvik バイトコードである．Dalvik バイトコードの生成は，Java プログラムをコンパイルした Java バイトコードより更に Dalvik バイトコードへ変換する．変換に用いるツールは dx と呼ばれ，AndroidSDK に含まれている．dx コマンドを用いて，クラスファイル（複数指定可能）を Java バイトコードに変換し，dex ファイルとして出力する．

Java プログラムをクラスファイル経由で Dalvik バイトコードで変換しているが，あくまでも現在の AndroidSDK の実装の制約であり，Java クラスファイルを経由する理由は特にない．今後 Java プログラムより直接 Dalvik バイトコードへコンパイル可能な開発ツールが登場する可能性は考えられる．

他のプログラミング言語から Dalvik バイトコードを出力する方法についても，すでに存在していたり，開発が進んでいる．その一例として Mono Project による MonoDroid[57] がある．Mono Project は .NET Framework 互換の開発環境，実行基盤として Mono を開発しているが，Mono Droid は Mono を Android に移植したものである．.NET Framework の中間言語である CLI（Common Language Interface） から Dalvik バイトコードへの変換するソフトウェアと， .Net 互換の API 群の移植により，例えば C# で Android アプリケーションの開発を可能としている．

4.3 dex ファイル

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・アクセスの削減を挙げている．そして，実行バイナリの dex ファイルはコンスタント・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，クラス毎にファイルが生成され，そこにクラス毎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 ファイルに纏められる形式を採っている．

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．

4.4 Dalvik バイトコードの仕様

Java バイトコードと同様に，Dalvik バイトコードもまた，命令は可変長である．**コードユニット**と呼ばれる 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．Dalvik バイトコードは計 218 命令存在する．

各命令は最大 8 つのオペランドを持つ．オペランドのうち，変数に相当するものは **Dalvik レジスタ** と呼ばれる．例として，図 4.1 に 3 つのバイトコードを示す．上から，2 バイト幅で 2 つの Dalvik レジスタ（vA，vB）をオペランドとして持つ `move` 命令，4 バイト幅で 3 つの Dalvik レジスタ（vA，vB，vC）をオペランドととして持つ `add-int` 命令，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として `move/16` 命令である．1 つのブロックは 1 バイトであり，`move/16` 命令は計 6 ブロック/バイトとなる．

命令によって，オペランドにて表現できる Dalvik レジスタのビット幅は異なる．例えば `add-int` 命令では，3 つ与えるオペランドについて，8 ビットの幅を持つため，Dalvik レジスタ 0 ~ 255 番の Dalvik レジスタを演算対象として表現できる，

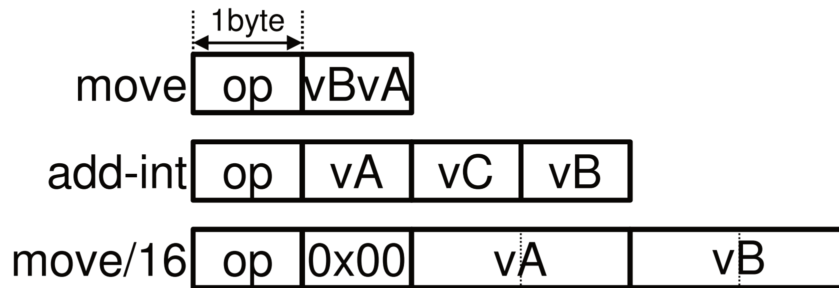

図 4.1: Dalvik バイトコードのフォーマット例

また，命令長を抑えるため，オペランドで表現可能な Dalvik レジスタのビット幅が異なる命令がいくつか存在する．例えば `move` 命令と，`move/16` 命令である．いずれも Dalvik レジスタ間の値を移動する命令であるが，オペランドで表現可能な Dalvik レジスタは，前者で 4 ビット，後者で 16 ビットとなる．アクセスできる Dalvik レジスタの範囲は，前者が 0 ~ 15，後者が 0 ~ 65, 536 であり，16 ビット幅のオペランドでなければ，すべての Dalvik レジスタへアクセスすることはできない．一方，命令長は，前者で 2 バイト，後者で 6 バイトとなる．オペランドで表現できる，Dalvik レジスタの範囲が異なる命令を複数設けることで，アクセスする Dalvik レジスタに適した短いバイトコードを選択することができる．

4.5 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することができる．ただし，Dalvik レジスタはメソッド内の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，AOSP にて公開されている標準アプリケーションでは，図 4.2 に示すように，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．最も多くの Dalvik レジスタを使用しているメソッドであっても 76 個である．プログラミングにおける実用上では，無尽蔵のレジスタを持つ VM とみてよい．各命令は，0 ~ 8 個の Dalvik レジスタをオペランドとし，それらの値を用いて演算，制御する．

ここで Dalvik レジスタは，プロセッサ上のハードウェア・レジスタと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 4 バイト幅の配列データに過ぎない．そのため，インタプリタが Dalvik レジスタを操作する際，例えば演算命令では

1. オペランドで指示された Dalvik レジスタの値を，物理レジスタへロードする．

2. 物理レジスタにロードした Dalvik レジスタ間で演算を行う．

3. 演算結果を，物理レジスタからオペランドで指示された Dalvik レジスタへストアする．

という処理が行われる．

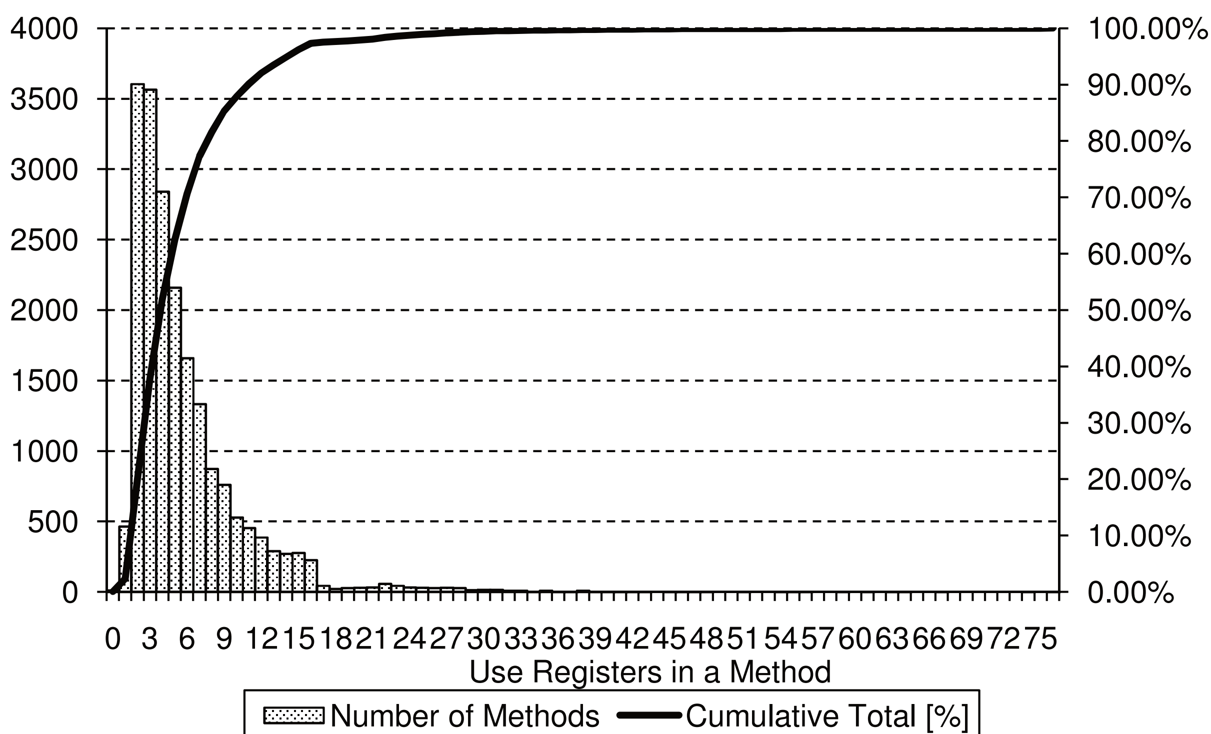

図 4.2: Android 標準アプリケーションにおけるメソッド毎の Dalvik レジスタ使用本数

Dalvik レジスタレジスタそのものは 16 ビット幅の値を持つが，コンパイル時に，メソッド単位で使用する Dalvik レジスタの個数が決定される．Java 同様にベリフィケーションが存在するためである．また，前述のとおり，命令によってオペランドで表現可能な Dalvik レジスタのビット幅は異なるため，アクセス頻度の高い変数，オブジェクトに，極力低い値の Dalvik レジスタをマップさせることが，バイトコードのサイズ削減につながる．

4.6 Dalvik VM の内部構造

Dalvik インタプリタが Dalvik バイトコードを実行する様子を図 4.3 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**

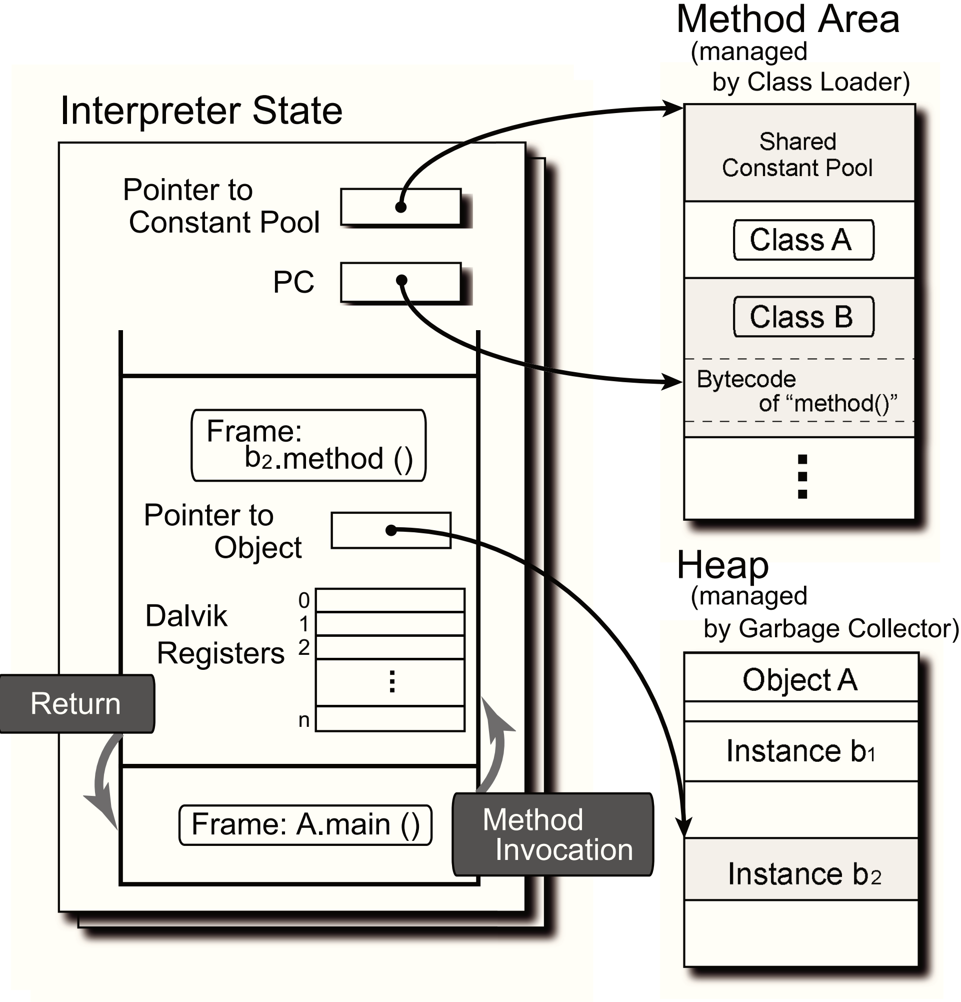

図 4.3: Dalvik インタプリタによる Dalvik バイトコード実行

4.6.1 メソッド領域

メソッド領域においては，4.3 で示したように，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．

4.6.2 ヒープ

ヒープ内部の構造，動作は Java VM と基本的には変わらない．ただし，Dalvik VM の GC は Mark & Sweep，Stop the World 方式の古典的な組み合わせを用いている [54].

Mark & Sweep 方式は，オブジェクトの参照を辿り，到達できなかったオブジェクトを破棄する手法である．不要なオブジェクトの破棄を安全に行えるが，動作速度は遅い．Stop the World 方式は，GC の発生時にすべてのスレッドを停止する手法である．GC 中はオブジェクトの参照に変化が発生しなくなるため，安

全にオブジェクトの破棄が行える一方，長時間の停止は応答性の低下を招く．高速ではない手法と，アプリケーション全体が動作を停止する手法の組み合わせから，Dalvik VM の GC 実装は芳しいものではない．

Android2.3 からは Stop the World 方式から，コンカレント GC に変更となった．コンカレント GC は処理の多くを，VM のスレッドと並列に行えるようにしたものである．Android2.3 の Dalvik VM では，GC による全スレッドの停止時間の目標を 3 ミリ秒としている [58].

4.6.3 Interpreter State

Runtime Data Area に替わり，Dalvik VM には Interpreter State と呼ばれる領域がある．スレッド毎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・スタック毎に指定していたコンスタント・プールは，クラスを問わず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．

4.7 Java VM との比較

4.7.1 スタック・マシンとレジスタ・マシン

Dalvik VM はレジスタ・マシンの VM であることから，バイトコード毎に演算対象となる Dalvik レジスタを指すオペランドが含まれる．よって，Dalvik バイトコードのバイトコード単位の命令長は Java バイトコードに比べて長くなる．

一方，Java バイトコードはバイトコードの演算対象，演算結果となる値，参照をオペランド・スタックへポップ/プッシュする操作が必要であるため，Dalvik バイトコードは命令数が Java バイトコードに比べて少なくなる傾向にある．

4.7.2 GC の方式

Java VM では，実行する環境や利用できる資源の大きさに適合した，効率の良い GC の手法が考案されてきた．例えば世代別 GC や コピー GC といった不要なオブジェクトを効率よく検出する手法，コンカレント GC などの停止時間を低減する手法である．

一方で，Dalvik VM では，述べたように GC の効率は高くない．1 回の GC で 100 ms 以上停止することもある [27]．アプリケーションのスレッドがすべて停止するため，UI スレッドも停止し，アプリケーショ

ンの応答性が低下，動作がぎこちなくなる．よって体感的な速度にも影響を及ぼす．特にゲームプログラミングにおいては，GC を極力発生させないようオブジェクト指向プログラミングを無視した設計が，テクニックとして挙げられてしまう程である [59].

4.7.3 格納ファイルの比較

表 4.1 に，3 種類の Android のコンポーネントのサイズを非圧縮の jar，圧縮した jar，非圧縮の dex 各形式で比較した内容を示す [20]．カッコ内は非圧縮 jar ファイルに対するサイズ比である．

Android の Java で記述された共有ライブラリにおいて，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 のサイズであるのに対し，dex ファイルは 10.3 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 ファイルは効率の高い実行バイナリの格納ができていることがわかる．

なお，Android のアプリケーション配布形式である .apk ファイルは，依存するリソースファイルと，dex ファイルをまとめた ZIP 形式である．

4.8 Android アプリケーションの動作

Android アプリケーションの実行は，まず zygote と呼ばれるプロセスから UNIX ベースのオペレーティングシステムのシステムコール fork を用いて，プロセスのコピーを生成するところから始まる．そして複製された Dalvik VM はアプリケーションの Dalvik バイトコードを読み込み実行を開始する．

zygote は Android のシステムの起動時に生成されるプロセスである．zygote は Dalvik VM や Dalvik バイトコードで記述されたクラス・ライブラリ，ネイティブ・コードによるライブラリをプリロードしメモリに配置する．よって多くのライブラリを事前に抱え込むこととなり，端末の起動には若干時間を要する．

この多数のライブラリを抱える zygote を fork システムコールにより複製することで，ライブラリなどすでにメモリ上にマップされたデータについて，親プロセスとアドレス空間を共有する．アプリケーションの起動毎にライブラリのロードを行わないことから，起動を高速にでき，個々のアプリケーションが使用するメモリ容量の増加を抑えられる．

表 4.1: dex ファイルと jar ファイルの効率比較

プログラム	非圧縮 jar バイト	圧縮 jar バイト (圧縮率)	非圧縮 dex バイト (圧縮率)
共有ライブラリ	21,445,320	10,662,048 (50%)	10,311,972 (48%)
Web ブラウザアプリ	470,312	232,065 (49%)	209,248 (44%)
アラーム時計アプリ	119,200	61,658 (52%)	53,020 (44%)

Android のアプリケーションは，各々固有のユーザ ID を持ち，独立したプロセスで動作する．そして，カーネルの持つセキュリティ･保護機構を利用することで，システムならびにアプリケーションを保護する．

メモリが許す限り，アプリケーションは複数起動される．一方，メモリが足りなくなった場合は，Low Memory Killer カーネルモジュールにより，強制終了され空きメモリ空間を確保する．

4.9 VM の高速化手法

Android2.1 までの Dalvik VM には，JIT や AOT といった，従来 Java で用いられてきた高速化手法は組み込まれていない．Android2.2 からは，トレースベースの JIT が Dalvik VM に組み込まれている．JIT に対応するアーキテクチャは，ARM と MIPS である．

インタプリタについては，C 言語で記述された汎用インタプリタと，アセンブラで記述された高速インタプリタが設けられている（更にデバッグ用のインタプリタも存在するが，割愛する）．対応する CPU アーキテクチャでは，後者を用いて動作速度を向上できる．アセンブラによるインタプリタは CPU 種別や浮動小数点ユニットの有無によって，別個に最適化されたバージョンが存在することもある．

アーキテクチャは共通だが，製品によって別個のインタプリタ用意されている例として，x86 アーキテクチャにおいて Intel Atom プロセッサに最適化されたインタプリタが挙げられる．

浮動小数点ユニット（FPU）の有無によって，別個のインタプリタが用意されている例として，ARM や MIPS アーキテクチャの Dalvik VM が挙げられる．これらのアーキテクチャでは，浮動小数点を扱うバイトコードのインタプリタコードにおいて，ソフトウェアによる浮動小数点演算コードではなく FPU による命令を用いることで動作速度を高めている．

第5章　Dalvik バイトコード・アクセラレータ

本章では，研究の対象である Dalvik バイトコード・アクセラレータのアーキテクチャについて示す．

5.1　Dalvik アクセラレータの実装方針

Dalvik アクセラレータでは Jazelle 方式を実装の手本として採用する．その背景として 2 点ある．

1. 節 3.3 ハードウェアによる方式 にて示したように，Jazelle DBX はほかの Java バイトコード・アクセラレータに比べて，ゲート数に対する性能向上，実装命令数の高さが特徴であり，必要なゲート数も少ない．よって，プロセッサに与える面積の影響は軽微にできる．独立したプロセッサ機能を有しゲート数が必然的に大きくなるコプロセッサ方式に比べると，その差は著しい．
2. 制御共有方式はプロセッサに組み込まれる特性上，プロセッサの再設計が必要にみられるが，大幅な変更にはならない． Jazelle DBX の実装で最も大きく占める個所は，Java バイトコードからプロセッサ・ネイティブなコードを生成するデコーダである．このデコーダステージは，プロセッサ・パイプラインの 1 つである．パイプライン化により，各ステージ毎に機能が細分化がされている．よってプロセッサの他のステージへの影響を少なくしながら，追加することができる．

Dalvik アクセラレータではターゲット・プロセッサのフェッチステージとデコードステージの間に，Dalvik バイトコードをネイティブ・コードへ変換するデコーダを設ける．デコーダにバイトコードが入力されると，VM が翻訳して実行する動作と相当の動作を行うネイティブ・コードを出力する．

5.2　適合するアーキテクチャ

Dalvik アクセラレータが適合しやすいアーキテクチャには，いくつかの要素が挙げられる．RISC アーキテクチャのプロセッサの方が，実装しやすい．

- 固定命令長である
  ターゲット・プロセッサが可変長命令セットの場合，生成するネイティブ・コードの中で最も長い命令長に合わせた出力バスを用意し，命令の長さを伝える仕組みも必要である．これは，デコーダの面積を増加させる要因となる．

- 演算サイクルが 1 サイクルである
  演算サイクルが均一でない場合，Dalvik デコーダが生成したネイティブ・コードの命令をデコードステージへ送り出す際，フロー制御が複雑になる．

- 演算はレジスタ間に限定
  演算命令から直接メモリへアクセスできれば，デコーダの生成するネイティブ・コードはメモリ上にある Dalvik レジスタに対して直接演算可能である．一方で，命令の動作にメモリアクセスと演算が分離していないため，実行サイクル数が均一になりにくく，前述の生成した命令のフロー制御が発生しやすくなる．
  また後述する，性能向上手法である DRMT 機構においては，RISC アーキテクチャのレジスタとメモリ間の転送命令が独立している特性を利用している．物理レジスタにロードされた Dalvik レジスタがどれかを記憶，参照できる仕組みを用いロード/ストアを削減している．

Android が公式にサポートするアーキテクチャは，ARM，MIPS，x86 とあるが，これらに適合するのは前者 2 アーキテクチャとなる．x86 アーキテクチャは，可変命令長であり，各演算命令から直接メモリ上のアドレスを指定可能な命令が存在する．したがって，Dalvik アクセラレータの実装が他アーキテクチャに比べて複雑となる．

5.3 プロセッサ上での構成

Dalvik アクセラレータの中心となる Dalvik デコーダは，Jazelle 方式同様に，プロセッサパイプライン中のフェッチステージとデコードステージの間に組み込まれる．フェッチステージからデータ列を取得し，デコードステージへ命令列を出力する．

図 5.1 に Dalvik デコーダを組み込んだプロセッサのブロック図を示す.

Dalvik アクセラレータは，次の機構からなる．

Dalvik デコーダ Dalvik バイトコードより変換を行い，ターゲット・アーキテクチャの命令列を出力する．

ステータスレジスタ 現在の実行モードを保持するレジスタ．ネイティブ・モードと Dalvik モードの 2 つのモードを識別する．

セレクタ ステータスレジスタの値より，デコードステージへ入力するデータを切り替える．ネイティブ・モードの場合は，フェッチステージより直接入力する．Dalvik モードの場合は，Dalvik アクセラレータより入力する．

5.4 モードの切替

ネイティブ・モードと Dalvik モード，2 つの実行モードの切り替えは，Jazelle DBX と同様に，プロセッサの既存部分に対し，モード切替を行う専用命令を実装して切り替えを実現する．

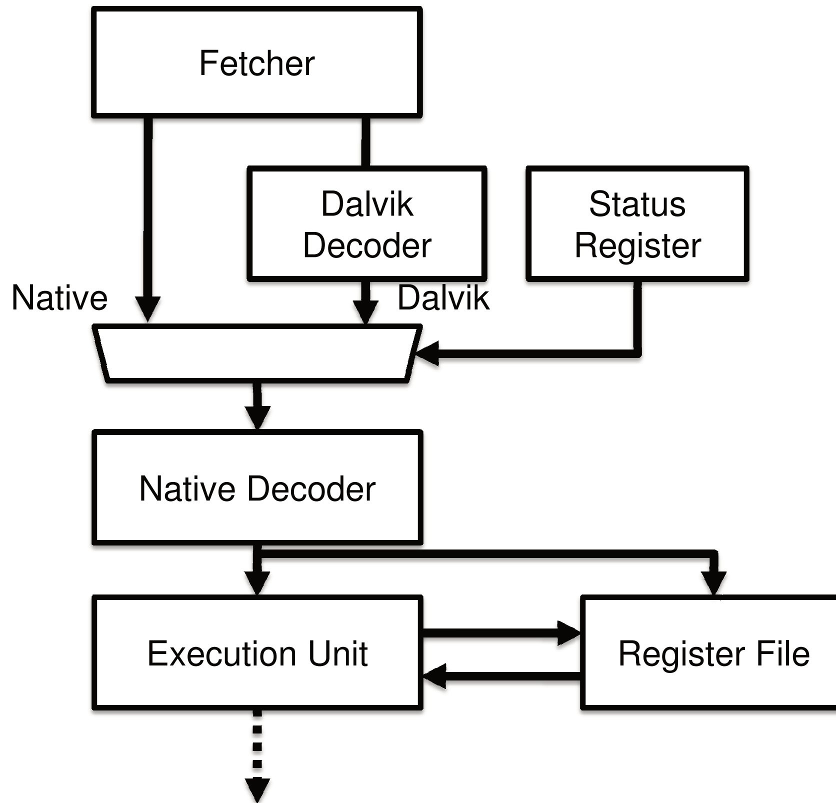

図 5.1: Dalvik デコーダを組み込んだプロセッサのブロック図

5.4.1　Dalvik モードへの切替

Dalvik モードへの遷移は，ターゲット・アーキテクチャのレジスタ分岐命令に Dalvik モードへの遷移を追加した命令を実装する．ここでは，`JRCD` (Jump Register and Change Dalvik bytecode mode) と呼ぶ．命令のオペランドはレジスタ絶対ジャンプ命令と同じで，宛先となるアドレスを指すレジスタを指定する．ここでは，アクセラレータで実行を開始したい Dalvik バイトコードのアドレスが設定された，レジスタを指定する．

JRCD 命令が実行されると，ステータス・レジスタが更新され，セレクタの入力が切り替わり，Dalvik デコーダを活性化する．

5.4.2　ネイティブモードへの切替

Dalvik モードからネイティブ・モードへ戻る命令も定義しなくてはならない．ここでは仮に `JRCM` (Jump Register and Change Mips mode) と呼ぶ．この命令をターゲット・プロセッサのデコードステージ，実行ステージに実装する．JRCM 命令が実行されると，ステータスレジスタを更新して，Dalvik デコーダが停止する．命令のオペランドには，ターゲット・アーキテクチャの命令がロードされた，実行を再開したいアドレスを含むレジスタを指定する．通常は，後述する VM のハンドラ・アドレスを含むレジスタを指定する．

JRCM 命令は Dalvik デコーダが使用する命令で，ユーザ・プログラムからは利用しない．プロセッサパイプライン上で，プロセッサ例外か Java 例外を検知する，もしくはデコーダが変換できない命令が Dalvik デコーダにフェッチされたとき，Dalvik デコーダは JRCM 命令を発行する．

なお，変換できない命令は，3.3.5 で述べた，Jazelle DBX が変換対象としない Java バイトコードと同等の機能を有する 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を含めて，**例外**と呼ぶこととする．

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（$ESTAT）へと書き込む．そして，上述の `JRCM` 命令によって，例外ハンドラへとジャンプする．なお，例外ハンドラ・アドレスは，後述する所定のレジスタ（$EHAND）に格納される．

ジャンプした先の例外ハンドラのコードでは，例外の要因を読み出して，それに応じた動作を行う．例として，扱えないバイトコードの場合は VM によるバイトコード実行を行ったり，Java 例外の場合は例外を catch するブロックの存在確認，プロセッサ例外であればプロセッサの例外処理へ移行する．

5.5 MIPS アーキテクチャにおける実装例

以下では，より詳細な実装を示す必要から，MIPS プロセッサ [60][61] [62] 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，以下で述べるアクセラレータの動作の大部分は，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，Dalvik アクセラレータによるバイトコードの変換方法，また，第 6 章で述べるメモリ・アクセスの削減方法は，Android にて広く用いられている，ARM アーキテクチャのプロセッサへアクセラレータを実装する場合にも適用できる．むしろ，後者の方法は，MIPS よりも ARM の方が効果が高い．詳しくは第 9 章で評価する．

5.6 レジスタの使用

3.3.5 で述べたように，Jazelle DBX では，ARM – Jazelle 間のモードのスイッチングを高速に行うために，VM が予約済みのレジスタ（3.1 の r6~r8，r15）は Java モードにおいて同様の目的で利用する．モード遷移時に値をロード・ストアするレジスタの本数を減らすことで，スイッチングを高速化する．Dalvik アクセラレータにおいても，Dalvik VM の機械語インタプリタが予約済みのレジスタの一部を Dalvik モード時に共用する．MIPS アーキテクチャの Dalvik VM のインタプリタにおいては，以下のレジスタが予約されている．

PC(r16) Dalvik インタプリタのプログラム・カウンタ．$INST と組み合わせて，分岐命令で再設定するプログラム・カウンタの値を計算，再設定する．

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．バイトコードよりデコードした命令が，Dalvik レジスタへアクセスするとき，本レジスタを相対アドレスとしてロード・ストアを行う．

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．通常，`JRCM` 命令の宛先オペランドとして用いられる．Dalvik モードに切り替える前に，ハンドラのアドレスを設定する必要がある．

EOBJ(r21) Java 例外の場合において，例外を起こしたオブジェクトへの参照が含まれる．

ESTAT(r22) 発生した例外の要因に対応した一意の値を収めたレジスタ．発生した例外の種類が格納される．例外ハンドラのコードは，このレジスタの値を読んでどのような理由で VM に制御が戻ったか識別する．

EPC(r23) 例外を起こしたバイトコードのアドレス．

また，デコーダがバイトコードより変換する MIPS 命令列が，一時的に使用するレジスタがある．$SCRH0~3 は，デコードした MIPS 命令列が一時的な演算のために用いるレジスタである．次のような目的で利用される．

表 5.1: MIPS アーキテクチャにおける Dalvik モード時のレジスタ割り当て

番号	名称	目的
r0	zero	常時 0 (MIPS の仕様 [62])
r1	at	マクロ命令が一時的に使用
r14-15	SCRH0/1	デコードした MIPS 命令が利用可能な一時レジスタ
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
r24-25	SCRH2/3	デコードした MIPS 命令が利用可能な一時レジスタ
r26-31	-	カーネル・MIPS プログラムが使用

- 複雑な演算を行う際の一時保存先.
- 配列へアクセスするために，添え字番号から求めだしたオフセットアドレスの計算結果.
- 桁上がり処理が必要な倍長演算の一時保存先.

これらのほかに，ゼロ・レジスタやオペレーティング・システムで予約されているとして，MIPS の規約上使用できない物理レジスタがある．これらのレジスタの使用状況をまとめた一覧を，表 5.1 に示す．表 5.1 に示す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，出力する MIPS 命令が Dalvik レジスタの値をロード・ストアするレジスタとして用いる．また，Dalvik アクセラレータに入力されるバイトコード間で，最近アクセスされた Dalvik レジスタを保持することで，Dalvik レジスタと物理レジスタ間のロード・ストアを削減する．詳しくは第 6 章で述べる．

5.7　アクセラレータの構造

アクセラレータは，大きく分けて，バイトコードバッファ，命令ジェネレータ，変換テーブル，命令テーブル，DRMT の 5 つのブロックからなる．図 5.2 に Dalvik デコーダの構造を示す．下記の項では，各ブロックの詳細を示す.

5.7.1　バイトコードバッファ

バイトコードバッファは，固定長の MIPS 命令から Dalvik バイトコードを形成するバッファである．本論文中で想定するアクセラレータの実装先である，MIPS アーキテクチャは，通常 32 ビット幅で命令データをフェッチする．

フェッチした 32 ビットのデータ列を 2 つの 16 ビットデータ列に分離し，下位 16 ビットを先に，上位 16 ビットを後にバッファへ投入する．16 ビット幅に分離することで，バッファ内では コードユニット毎に格納される．最も長い Dalvik バイトコードで長さは 5 コードユニット，80 ビットであることから，最低でも 5 つの コードユニットをバッファ可能な構造でなければならない．

Dalvik バイトコードは 4.4 で示すように，先頭のコードユニットの下位 8 ビット，1 バイト目が OP コードのフィールドとなっている．バイトコードバッファは，バッファの先頭の該当箇所を参照し OP コードを識別する．バイトコードバッファ内部には，OP コードでインデックシングされた，コードユニット長を格納したテーブル**コードユニット長テーブル**を持つ．バッファに積まれたコードユニットの個数と比較することで，バイトコードのフェッチ完了を確認する．

図 5.3 にフェッチステージより，バイトコードバッファにデータ列が積まれた状態の例を示す．ここでは，フェッチステージより 32 ビットのデータ列 `0x016A1002` が入力されたものとする．バイトコードバッファは，キューの先頭の下位バイトを参照し，オペコードを参照する．ここでは，0x02 を参照することで，キューの先頭にあるコードユニットからバイトコードを形成するために必要な，コードユニット長は 2 で

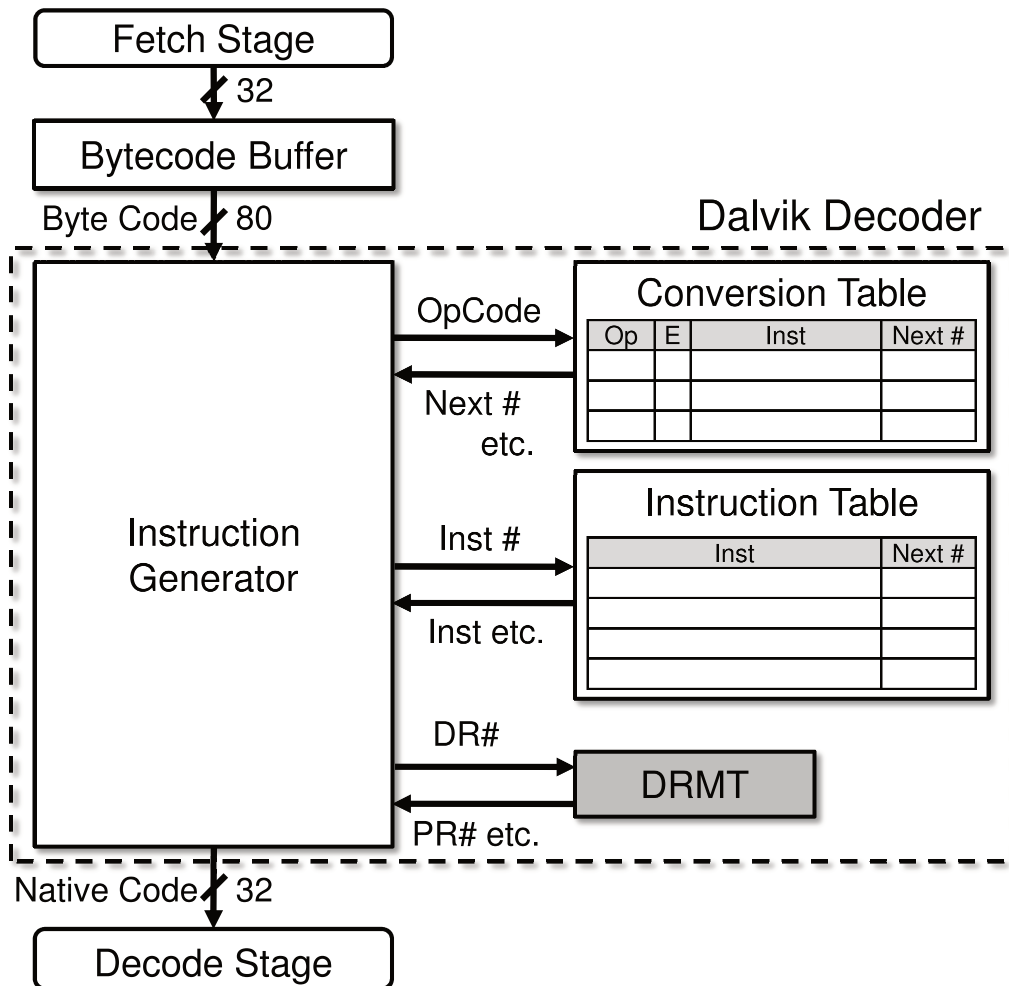

図 5.2: Dalvik デコーダのブロック図

あることがわかる．キューにはコードユニットがすでに 2 つ入力されているので，バイトコードを形成できる状態であることがわかる．

バイトコードを形成できる状態になったら，図 5.3 に示すようにキューからバイトコード出力にデータを移動する．そして，Ready フラグを有効にし，命令ジェネレータがバイトコードを取得してよいことを通知する．命令ジェネレータはこの Ready フラグを基に，バイトコードバッファからバイトコード列をフェッチしてよいか確認する．バイトコードを取得したら，Ready は再び無効となる．

コードユニット長テーブルにおいて，未定義のバイトコードのコードユニット長は 0 とする．バッファに積まれたコードユニットに対応する，バイトコードのコードユニット長が 0 の場合，バイトコードバッファは未定義命令例外を発生する．

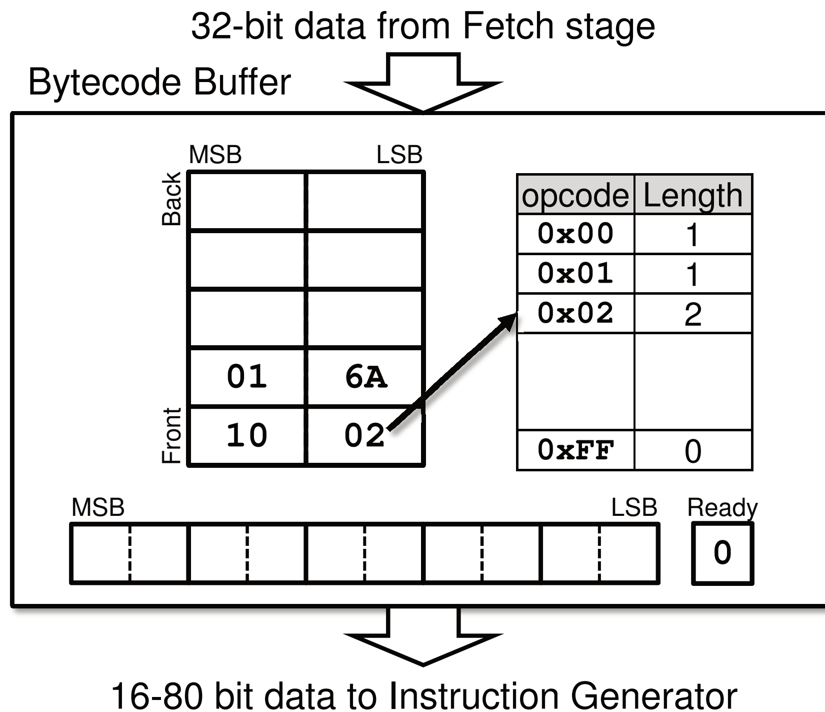

図 5.3: バイトコードバッファ 積まれた状態

5.7.2　命令ジェネレータ

命令ジェネレータは，入力された Dalvik 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である．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．

変換テーブル，命令テーブルからは，出力する MIPS 命令列の元となるひな形を読み出す．そして，バイトコードから得られたオペランドをこのひな形にセットすることで，MIPS 命令列を生成する．生成した MIPS 命令列は，後続のユニットへ逐次発行する．発行された MIPS 命令を，後続のユニットが取得したら次の命令の発行を続ける動作を行う．

5.7.3　変換テーブル

変換テーブルは，アクセラレート可能なバイトコードの命令列を保持する．変換テーブルは次のフィールドから構成される．

Op Dalvik バイトコードの OP コード．変換テーブルは OP コードによりインデクシングされる．よって，エントリ数は 256 となる．

E デコーダが変換可能か示すフラグ．デコード可能なバイトコードの場合，このフィールドの値がアサートされる．

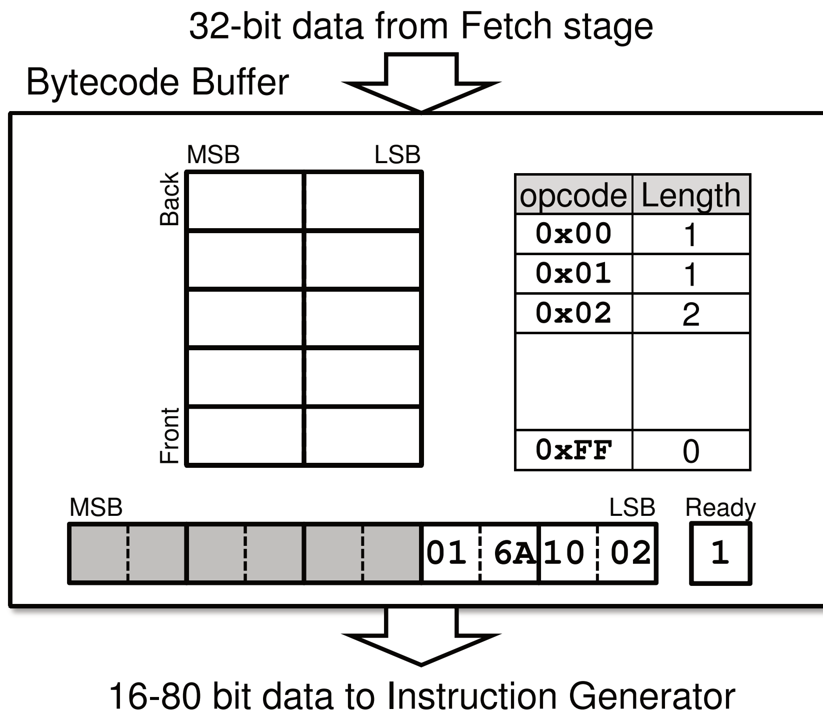

図 5.4: バイトコードバッファ バイトコードを形成

Inst Dalvik バイトコードに対応する，MIPS 命令のひな形ならびに生成に必要な情報を収める．命令テーブルの参照による命令発行のレイテンシ増加を抑えるため，命令テーブル上の対応する Inst フィールドの内容を先頭から数命令キャッシュする．

Next # 後続して生成する MIPS 命令列を収めた，命令テーブルの対応するエントリへのポインタ．バイトコードからデコードする MIPS 命令が複数に及ぶ場合，後続して発行する MIPS 命令の形成に必要な情報がどこにあるか，このフィールドを参照して辿る．

5.7.4 命令テーブル

命令テーブルは，バイトコードからデコードされる，2 番目以後の MIPS 命令列を保持したテーブル．命令テーブルは以下のフィールドから構成される．

Inst MIPS 命令ならびに生成に必要な情報を収める．

Next # 次に出力する MIPS 命令列に対応するエントリを収めたポインタ．デコードする命令列が，複数の MIPS 命令となる場合，このポインタを辿り，順次命令を出力する．ポインタが指し示す先がない場合，バイトコードから出力される MIPS 命令列はなく，デコードを終了する．

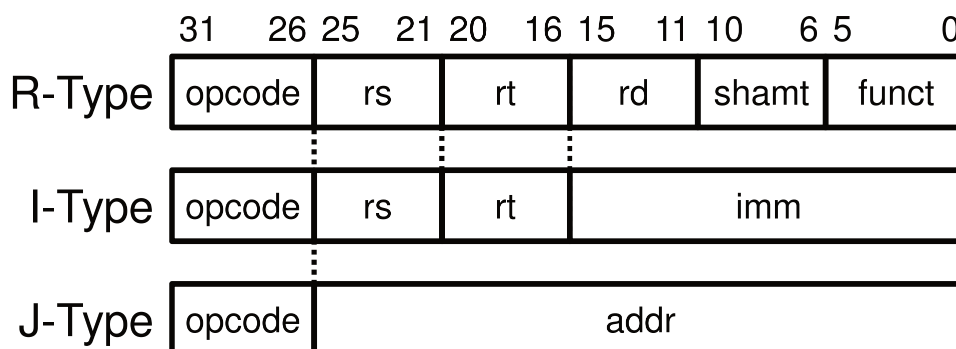

図 5.5: MIPS 命令セットのフォーマット

5.7.4.1 Inst フィールドの内容

変換テーブルならびに命令テーブルの Inst フィールドは更に，複数のフィールドから構成される．各フィールドは，出力する MIPS 命令列の opcode や funct フィールドの値，各フィールドの値の設定方法，どこから取得するのかについて定義される．以下に主なフィールドの内容を示す.

insttype 生成する MIPS 命令のフィールドタイプを指定する．MIPS 命令は 3 種類の命令フォーマットが存在し，それぞれ R 形式，I 形式，J 形式と呼ばれる．現在の Dalvik アクセラレータの設計では，J 形式は使用されない．これにより，以下のうち使用するフィールドが決定される．

opcode opcode フィールドにセットする値.

funct R 形式の命令にて，funct フィールドにセットする値.

shamt シフト量．R 形式命令にて用いられる．

rs，**rt**，**rd** 各フィールドは物理レジスタ番号を指す．指定できるレジスタ番号は，Dalvik レジスタにマップされる領域，一時レジスタ，VM ならびにアクセラレータ向けに予約されたレジスタとなる．

imm 即値フィールド．R 命令にて用いられる．取得方法は 2 種類ある．一つはバイトコード列の特定のビットフィールドから Dalvik レジスタ番号を取得する．Dalvik レジスタのロード・ストアを行う命令においては，オフセットアドレスを計算するために用いる．もう一つは固定値である．

図 5.5 に MIPS 命令の各形式のフォーマットを示す．各フィールドに，Inst フィールドの各要素で指示された値，もしくはバイトコードの特定のビット範囲より取得した値がセットされ，MIPS 命令を形成する．

5.7.5 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．これによりロード・ストア命令を削減する．その動作と内部構造については，第 6 章にて示す.

5.8　プロセッサに追加する命令

5.4 にて，アクセラレータのモード切替に関する命令を示したが，更にアクセラレータの動作を制御する追加命令が含まれる．例外が発生する可能性のあるオペランド値の検知や，分岐の成否によるデコード命令のフラッシュといった制御を行う命令が追加される．

これらの命令は，通常の MIPS モードでは使用されず，実行する MIPS アーキテクチャのバイナリにも含まれない，Dalvik アクセラレータ専用の命令である．MIPS 命令セットは，opcode ならびに funct フィールドの値により命令を決定するが，命令が定義されていない各々の値に，以下で示す命令を割り振る．

5.8.1　例外チェック命令

Dalvik アクセラレータが扱える Dalvik バイトコードの中には，例外を発生させる可能性のあるバイトコードが存在する．例外が発生する条件を検知し，必要であれば例外を発行する命令を設ける．挙げる例外はそれぞれ，アクセラレータから戻る際の要因を格納するレジスタ ESTAT に格納する値が定義されている．各チェック命令の条件を満たした場合，例外に対応する値を $ESTAT にセット，$EPC に例外の発生した PC をセット，そして JRCM 命令を発行し MIPS モードに切り替える．

戻った先のハンドラ・ルーチンにて $ESTAT の値を読み出し，発生した例外を識別し，適切な動作を継続する．

ヌル参照チェック　命令フォーマット: `EXC-NULL rs`

オペランド rs で指示されたレジスタ値がゼロならば，NullPointerException 例外の発生とみなす．配列などの参照が含まれた Dalvik レジスタの値が，ゼロ，すなわち空参照でないかチェックするのに用いる．

ゼロ除算チェック　命令フォーマット: `EXC-AE rs`

オペランド rs で指示されたレジスタの値がゼロならば，ArithmeticException 例外の発生とみなす．除算を行うバイトコードにおいて，除数が収められた Dalvik レジスタの値がゼロでないかチェックするのに用いる．

配列の添え字番号あふれ　命令フォーマット: `EXC-AIOB rs，rt`

オペランド rs に添え字番号の収められた Dalvik レジスタの値，オペランド rt に配列の要素番号の収められた Dalvik レジスタの値が含まれた物理レジスタを指示する．$rs > rt$，すなわちアクセスしようとする添え字番号が要素数を上回ってい場合， ArrayIndexOutOfBoundsException 例外の発生とみなす.

第 2 オペランドの要素数は，アクセスしようとする配列の参照を経由して取得する．よって，配列へアクセスするバイトコードでは，`EXC-NULL` 命令を先にデコード・実行してから，配列の参照が空でないかチェックしてから `EXC-AIOB` 命令をデコード・実行する必要がある．

5.8.2　分岐用パイプライン制御命令

命令フォーマット: AC-FLS-EZ rs，AC-FLS-NZ rs

オペランド rs で指定されたレジスタの値を参照し，AC-FLS-EZ はゼロの場合，AC-FLS-NZ は非ゼロの場合，後続する，同一のバイトコードからデコードされた MIPS 命令を削除する．命令ジェネレータが同一のバイトコードより命令を発行している場合は，その命令のデコードも中断する．

条件分岐を行うバイトコードから出力される MIPS 命令列は，分岐の可否の判定に続き，MIPS 命令列の最後にプログラム・カウンタを更新する MIPS 命令列が出力される．この分岐の可否の判定を行うのが AC-FLS-EZ /AC-FLS-NZ 命令である．

出力される MIPS 命令には，入力される バイトコード単位で一意な バイトコード ID と呼ばれる情報を持つ．AC-FLS 命令は条件を満たすとき，自らのバイトコード ID と等しい，後続の MIPS 命令を NOP 命令に置き換え，無効にする．そして，命令ジェネレータがデコード中のバイトコードのバイトコード ID が等しい場合，命令の発行を中断させ，次のバイトコードの入力の受付を開始する．

分岐命令の条件が成立する場合は，出力された MIPS 命令列を順次実行し，プログラム・カウンタを更新する MIPS 命令が実行される．一方，不成立の場合は，プログラム・カウンタを更新する MIPS 命令の実行を抑止する．このときアドレスが更新されないので，次のアドレスのバイトコードから変換された MIPS 命令列が実行される．

図 5.6 に分岐命令時のプロセッサ・パイプラインの状態を示す．ここでは，後述する Dalvik アクセラレータの実装先プロセッサの 7 段パイプラインにて示す．

最上位にある I_0 は，分岐命令の実行に差し掛かっている状態である．MIPS 命令を実行する EX ステージに分岐判定命令，ここでは AC-FLS-EZ /AC-FLS-NZ が含まれる．分岐命令で成立した場合 Taken 側 I_{1A}，I_{2A} と動作する．一方，不成立の場合は Untaken 側 I_{1B}， I_{2B} と動作する．

各ステージの命令の上に記載されている値は，説明をわかりやすくするため，設けた仮のバイトコード ID である．分岐命令のバイトコードからデコードされた MIPS 命令列のバイトコード ID は 1，分岐命令の次のアドレスにあるバイトコードからデコードされた MIPS 命令列のバイトコード ID は 2，分岐した先のバイトコードからデコードされた MIPS 命令列のバイトコード ID は 3 とする．

まず，分岐の判定前の動作に注目する．I_0 のとき，EX ステージに分岐判定命令，その直後 RF ステージには PC を更新する MIPS 命令が投入されている．更に連続するアドレスのバイトコードより，デコードされた MIPS 命令列が ID，IF ステージにすでに投入されている．

条件分岐が成立すると，I_{1A} へと状態が移る．分岐条件が成立したと判定されると，AC-FLS-EZ /AC-FLS-NZ 命令は後続する自らと同じバイトコード ID を持つ命令のフラッシュ動作が行われない．よって，EX ステージにある，後続するプログラム・カウンタを更新する命令が発行される．

すると，I_{2A} 状態に示すように，命令ジェネレータがこれを検知し，プロセッサ・パイプラインをフラッシュする動作に入る．これによりバイトコード ID が 2，投機的に投入された MIPS 命令列がフラッシュされ，このバイトコードのデコードが中止される．そして，新しく設定されたプログラム・カウンタのアドレスよりバイトコードをフェッチ，デコードを再開する．

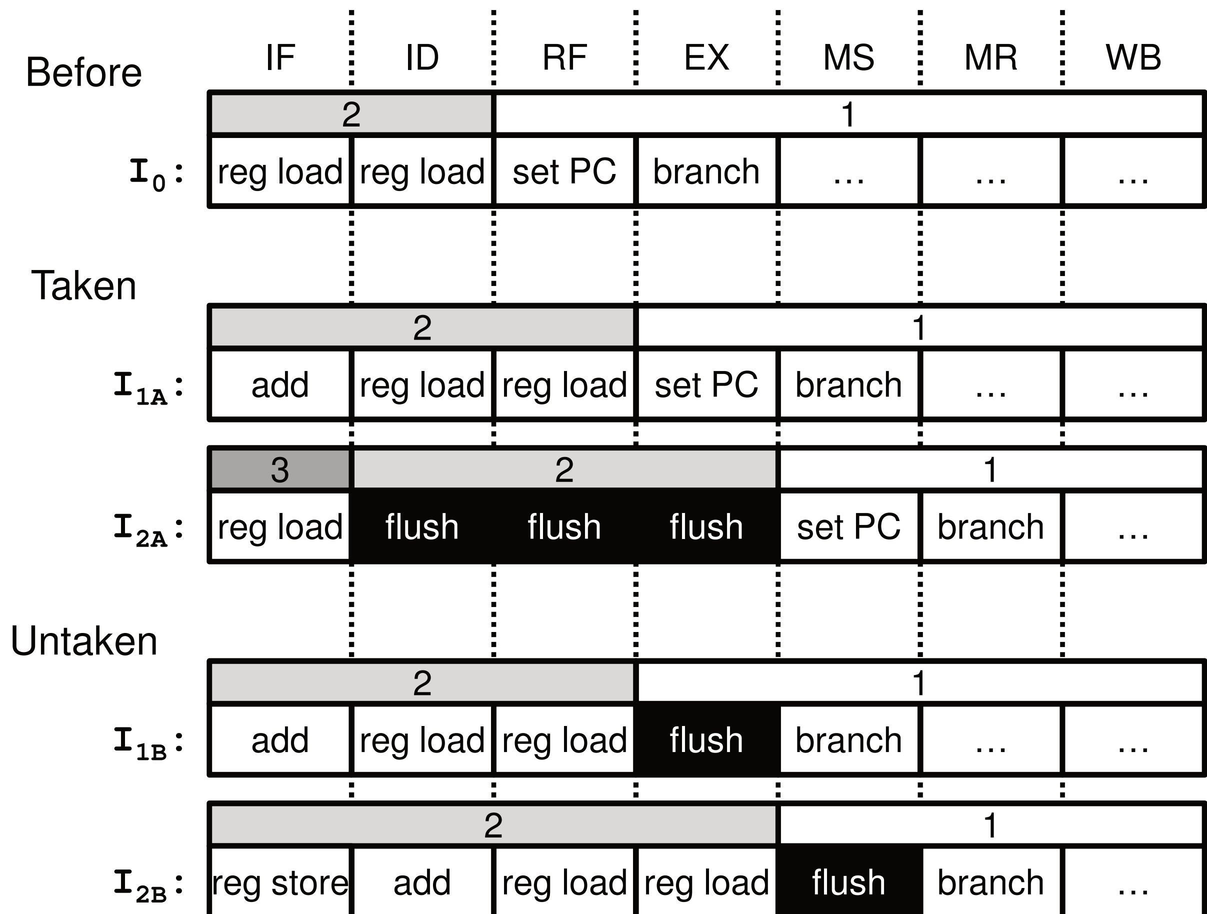

図 5.6: 分岐命令の動作

条件分岐が成立しない場合，I_{1B} へと状態が移る．分岐命令が成立しないと判定されると，`AC-FLS-EZ` `/AC-FLS-NZ` 命令は後続する自らと同じバイトコード ID を持つ命令のフラッシュ動作が行われる．これにより，プログラム・カウンタを更新する MIPS 命令はフラッシュされ実行されなくなる．

そして I_{2B} 状態に示すように，すでにパイプラインに投機的に投入されている，分岐命令に後続する，バイトコードよりデコードされた MIPS 命令列には影響は及ぼさない．

5.9 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 バイトコードよりデコードする基本動作の例として，Dalvik レジスタ間で加算演算を行う，`add-int`
- 一時レジスタを利用し，倍長の加算演算を行う `add-long/2addr`
- 例外の発生の可能性がある，配列への書き込みを行う `aput`

これらのバイトコードから，MIPS 命令列が発行される場合の動作について示す.

5.9.1 add-int 命令の変換

`add-int v0, v1, v2` は，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 5.1 の r17）を表す.

1. `lw $2, 4($FP)`
2. `lw $3, 8($FP)`
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 5.7 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 である．この値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，MIPS 命令列への変換の可否，最初に出力する MIPS 命令のひな形，次の参照すべき命令テーブルのポインタを取り出す．E フィールドの値より，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わかる．そこで，命令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 2 番であることを取得する．命令ジェネレータは，Dalvik レジスタ 2 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．Dalvik レジスタは 32 ビット幅であるため，Dalvik レジスタポインタからのオフセットアドレスを示す，即値の値は Dalvik レジスタ番号の 4 倍の値となる．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のに必要な処理を行い，`JRCM` 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 MIPS 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．取り出したひな形より，出力する MIPS 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．lw 命令の即値には，Dalvik レジスタ番号から計算された 8 がセットされる.
4. 繰り返し，次の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. 取り出したひな形と宛先オペランドを組み合わせ，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．即値には 0 がセットされる．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．よって，次のバイトコードをフェッチする.

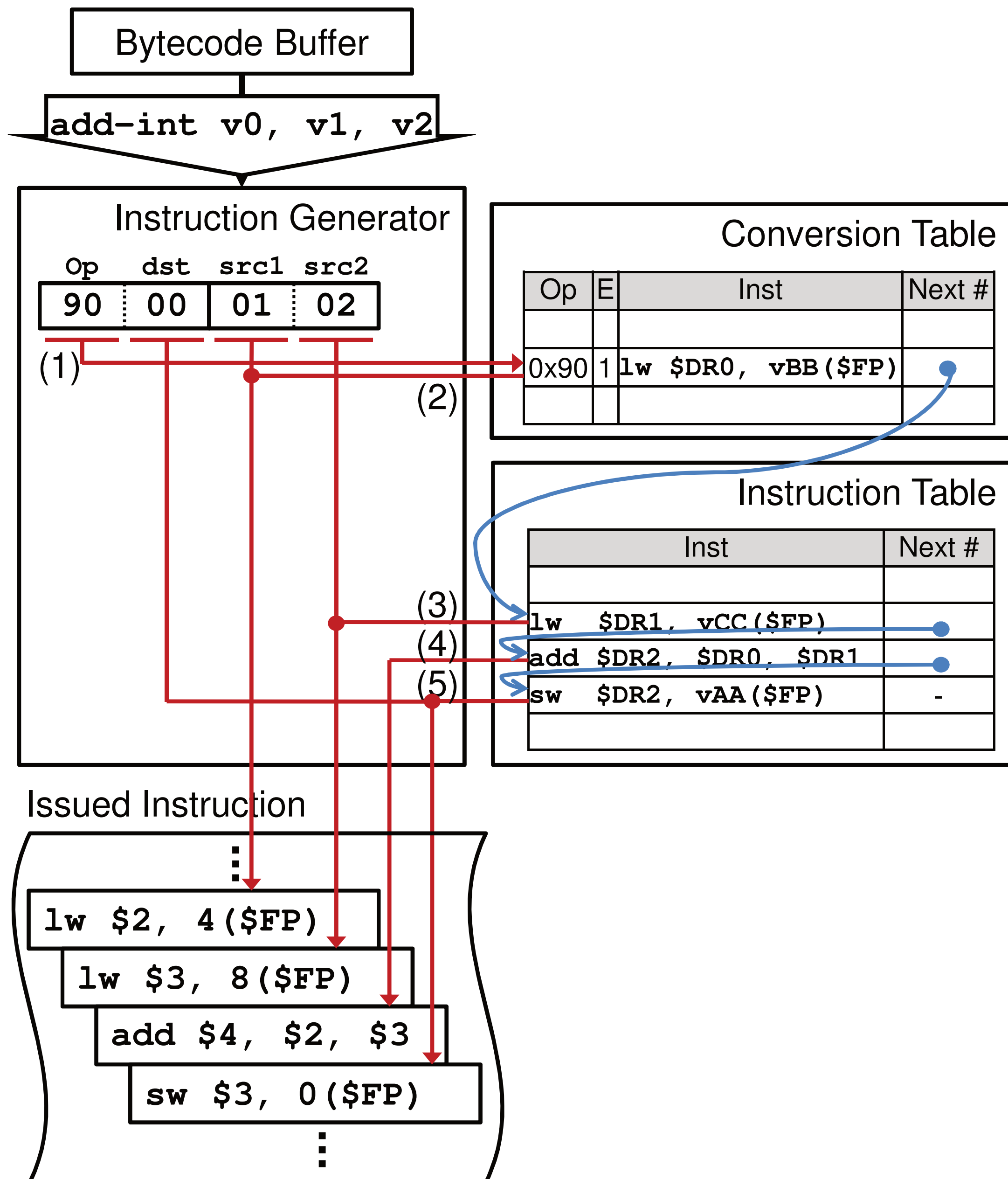

図 5.7: add-int 命令の変換例

5.9.2 add-long/2addr 命令の変換

次に，一時レジスタを利用する倍長演算命令のデコード例について示す．`add-long/2addr v0, v2, v4` は，Dalvik レジスタ 2 番と 4 番の間で加算，Dalvik レジスタ 0 番へ加算結果を書き込む．Dalvik VM において，倍長のデータ型である long 型， double 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．変換した結果より，次の 10 つの MIPS 命令列が出力される．ただし，$x ($x = 2 \sim 8, FP, SCRH0 \sim 1$) は物理レジスタを表すものとする．$SCRH0 ~ 1 は，デコードした MIPS 命令

列が使用可能な一時レジスタである．

1. `lw $2, 8($FP)`
2. `lw $3, 12($FP)`
3. `lw $4, 16($FP)`
4. `lw $5, 20($FP)`
5. `addu $6, $2, $4`
6. `addu $SCRH0, $3, $5`
7. `sltu $SCRH1, $6, $4`
8. `addu $7, $SCRH0, $SCRH1`
9. `sw $6, 0($FP)`
10. `sw $7, 4($FP)`

アクセラレート可能かつ複雑な処理の伴う Dalvik バイトコードの中には，演算の過程において，一時レジスタ ($SCRH0 ~ 3) を利用する MIPS 命令列を出力する場合がある．一時レジスタは，バイトコードからデコードされる MIPS 命令列単位で利用可能なレジスタである．add-long/2addr 命令においては，一時レジスタを，下位 32 ビットにて発生した桁上がりの検知と，その結果を上位 32 ビットに加算する動作の過程で利用している．

1 ~ 4 命令目までは，Dalvik レジスタの物理レジスタへのロード，9，10 命令目は物理レジスタから Dalvik レジスタへのストアを行っている．前述のとおり，倍長のデータ型における演算は連続した 2 つの Dalvik レジスタを 64 ビットの値として扱う．よって，各オペランドで指した Dalvik レジスタと，更に次の Dalvik レジスタをセットにしてロード・ストアしている．オペランドで指した Dalvik レジスタが下位 32 ビット，次の Dalvik レジスタが上位 32 ビットとなる．Dalvik レジスタ 2，3 番のペアと 4，5 番のペア間で加算演算を行い，その結果を 0，1 番のペアへストアする．

6 命令目において，上位 32 ビットの加算結果をいったん $SCRH0 に保持，7 命令目において桁上がりの検知を行いその結果を $SCRH1 に格納している．そして 8 命令目において，上位 32 ビットのみで加算した結果と桁上がりの検知結果を加算し，上位 32 ビットの値を決定する．

このように，Dalvik バイトコードからデコードされる MIPS 命令列単位で利用可能な，一時レジスタを設けることで，より複雑な処理を行うバイトコードのうち，アクセラレート可能なバイトコードの率を向上させることができる．一時レジスタはデコードされるバイトコード単位で利用可能であり，バイトコードからデコードした MIPS 命令列がプロセッサ・パイプラインからコミットされたら，次のバイトコードからデコードされた MIPS 命令列が一時レジスタを利用できる．

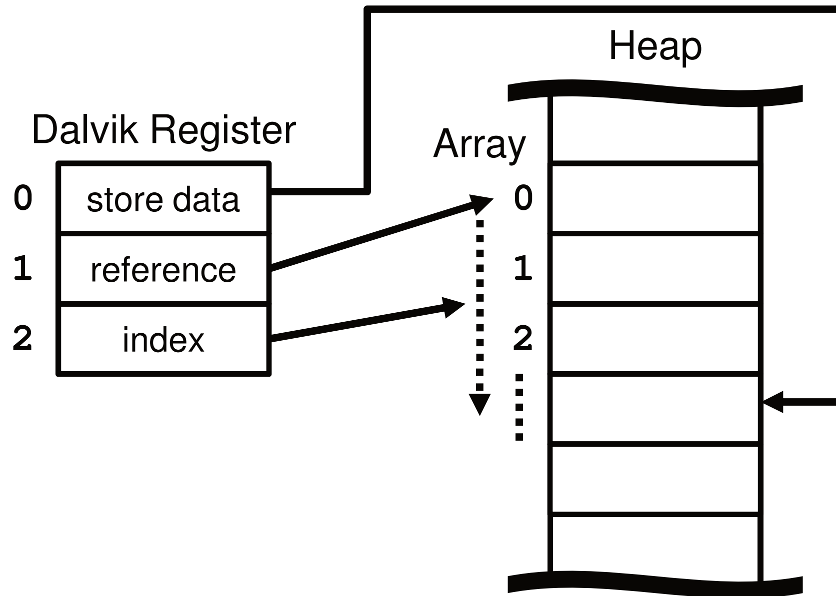

図 5.8: aput-int 命令の動作模式図

5.9.3 aput-int 命令の変換

次に，例外の検知例として配列操作命令のデコード例を示す．`aput-int v0, v1, v2` は，Dalvik レジスタ 1 番に格納された配列 (の参照) より，Dalvik レジスタ 2 番に格納された要素番号へアクセスする．そして Dalvik レジスタ 0 番の値を配列へストアする命令である．図 5.8 に動作の模式図示す.

配列を操作する命令においては，2 つの例外の検知を行わなければならない．一つは配列の参照が，空でないかである．空の場合は，NullPointerException 例外を発生させる必要がある．もう一つはアクセスしようとしている添え字番号が，配列の要素数を上回っていないかである．上回っている場合は，アクセスしようとする要素番号が不正であることを示す ArrayIndexOutOfBoundsException 例外を発生させる必要がある．

変換した結果より，次の 9 つの MIPS 命令列が出力される．ただし，$x ($x = 2 \sim 4, FP, SCRH0 \sim 1$) は物理レジスタを表すものとする．$SCRH0 ~ 1 は，デコードした MIPS 命令列が使用可能な一時レジスタである．

1. `lw $2, vBB($FP)`
2. `lw $3, vCC($FP)`
3. `EXC-NULL $2`
4. `lw $SCRH1, 8($2)`
5. `EXC-AIOB $3, $SCRH1`
6. `sll $SCRH0, $3, 2`

7. `addu $SCRH1, $2, $SCRH0`

8. `lw $4, vAA($FP)`

9. `sw $4, 12($SCRH1)`

1，2 命令目で，ストアの対象となる，配列の参照と添え字番号を取得する．続いて，EXC-NULL 命令により，ロードした配列の参照が 0，すなわちヌル参照でないかをチェックする．ヌル参照であった場合は，NullPointerException 例外の発生を理由として MIPS モードに実行モードを戻す.

4 命令目では，配列の参照を経由して，配列の要素数を取得する．Dalvik VM では，配列の参照アドレスから，8 バイト目に配列の要素数が格納されている．ロードした配列の参照から 8 バイトのオフセットを与えた先から，一時レジスタ $SCRH1 へ値をロードする．

続いて 5 命令目では，EXC-AIOB 命令により，アクセスする添え字番号 ($3) が配列の要素番号 ($SCRH1) を上回っていないか確認する．もし上回っている場合は，ArrayIndexOutOfBoundsException 例外の発生を理由として，MIPS モードへ実行モードを戻す.

発生する可能性のある例外を確認したら，配列の実体へアクセスする．6 命令目では，アクセスする添え字番号をビットシフト，要素の大きさで乗算を行い，オフセットアドレスを $SCRH0 に保存する．そして 7 命令目では，オフセットアドレスと配列の参照との間で加算を行い，アクセスするアドレスを決定する.

続いて，オペランド vAA で示された，ストアする Dalvik レジスタの値を $4 へロードする．最後にロードした Dalvik レジスタの値を，アドレスの計算を行った指定の配列の要素へ書き込む．このとき，sw 命令のイミディエイト値に 12 を指しているのは，配列のデータ本体が参照から 12 バイト目より存在するからである．

このように，デコード可能なバイトコードのうち，例外の発生する可能性のある命令は，事前にチェックを行う命令が出力される．MIPS 命令ではカバーできない，Java 例外の発生処理ついては，命令を追加することで対応する．

5.10　まとめ

本章では，Dalvik バイトコード・アクセラレータの仕様と動作について示した．

まず，どのようにアクセラレータがプロセッサに組み込まれるかについて，既存のプロセッサ・パイプラインに並列に，Dalvik バイトコードを入力するパイプラインを設けていることを示した．そして既存のプロセッサが扱う機械語と，Dalvik バイトコードとの間で切り替えることを示した．切り替えは Jazelle DBX 同様に，モードを切り替える命令により切り替え，これらの命令がどのように用いられるかについて示した.

そしてアクセラレータの内部構造と各機能の役割について紹介し，Dalvik バイトコードより MIPS 命令列が出力されることについて示した．より多くの複雑なバイトコードをデコードするための手法として，一時レジスタを使用することを挙げた．そして例外が発生する可能性のあるバイトコードについては，そのチェックを行う MIPS 命令列を出力し，更に所定の条件が成立したとき，実行モードを戻す命令の追加によって実現する．

第6章　DRMT

本章では，Dalvik アクセラレータより出力される MIPS 命令列について，Dalvik レジスタのロード・ストアにおける冗長なアクセスの存在について示す．そして冗長なメモリアクセスを削減する仕組みとして，Dalvik Register Map Table (DRMT) を挙げ，その動作と効果を示す．

6.1　Dalvik レジスタの冗長なロード・ストアの存在

第 4 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/ディ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 VM の 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，ソースとなる Dalvik レジスタの値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となる Dalvik レジスタへ書き込む際は，その都度，対応するアドレスへ演算結果をストアする．

図 6.1 に冗長なロード・ストアの例として，`add-int v0, v1, v2` と `sub-int v3, v0, v2`，2 つのバイトコードからデコードされた MIPS 命令列を例に示す．`add-int v0, v1, v2` からは，Dalvik レジスタ 1 番と 2 番からロードを行い，加算結果を Dalvik レジスタ 3 番へストアする MIPS 命令列が出力されている．`sub-int v3, v0, v2` からは，Dalvik レジスタ 0 番と 2 番からロードを行い，減算結果を Dalvik レジスタ 3 番へストアする MIPS 命令列が出力されている．

このとき，Dalvik レジスタ 0 番について注目すると，add-int 命令の最後に sw 命令によりストアを行っているが，その直後，sub-int 命令の最初で lw 命令によりロードを行っている．このような動作を **read after write** という．add-int 命令の時点で，Dalvik レジスタ 0 番の内容は $4 に存在している．sw 命令を用い同一のメモリアドレスからロード・ストアを行い値を受け渡すのは効率が悪い．

次に，Dalvik レジスタ 2 番について注目すると，add-int 命令の 2 番目にて lw 命令により $3 へロードされている．そして sub-int 命令の 2 番目にて再び lw 命令により，$3 へロードを行っている．Dalvik レジスタ 2 番ならびに $3 の値は，2 回目のロードのときまで値が変更されていない．よって，2 度ロードする必要はなく，効率が悪い．このような動作を **read after read** という．2 度ロードするのではなく，1 回目にロードした $3 を再利用すべきである．

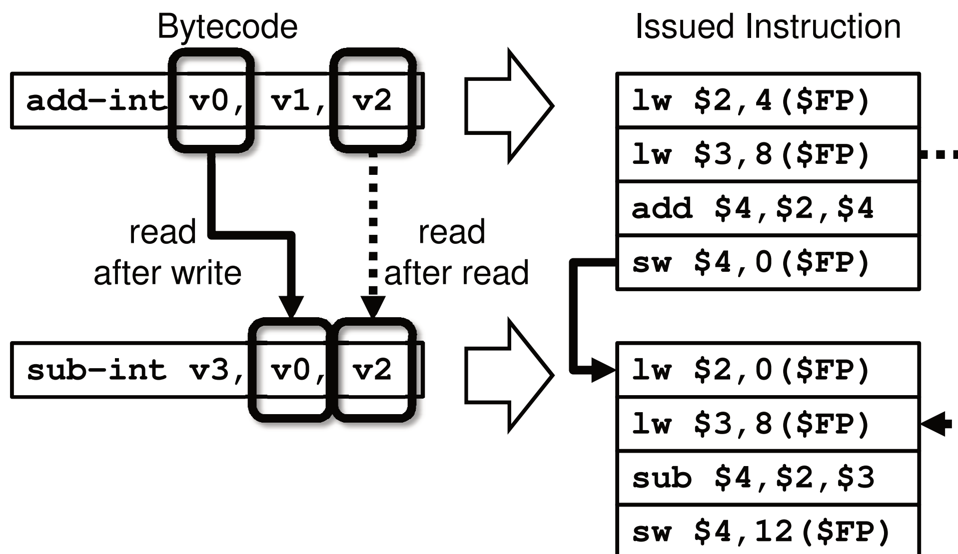

図 6.1: 冗長なロード・ストアを含むバイトコードからデコードされた MIPS 命令列の組み合わせ

このような冗長なロード・ストアを排除した，理想的なロードストアを図 6.2 に示す．まず，read after write ならびに read after read なロード・ストアとなった，sub-int 命令の 1，2 番目の 2 つの lw 命令は省略できる．

また，両バイトコードにおいて結果を Dalvik レジスタへストアする sw 命令についても，即座に Dalvik レジスタへ送る必要がない．後続するバイトコードにて，同一の Dalvik レジスタに対し書き込みが行われる可能性ある．もしその都度 Dalvik レジスタへストアを行っては，効率が悪い．

Dalvik レジスタのストア動作は物理レジスタの値の更新に留めておき，ほかの Dalvik レジスタのロードや実行モード遷移により物理レジスタを解放しなければならない時に，値の更新されている Dalvik レジスタをストアするよう，ストア動作を限定すればストア命令の効率は高まる．

6.2 Dalvik レジスタのロード/ストアを削減する手法の考察

このようなロード/ストア動作について，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．

まず，考えられる Dalvik レジスタの物理レジスタへのマップ手法の例を挙げ，その問題点を示す．その上で Dalvik バイトコード・アクセラレータに搭載する，Dalvik レジスタの物理レジスタへのマップ手法を示す．

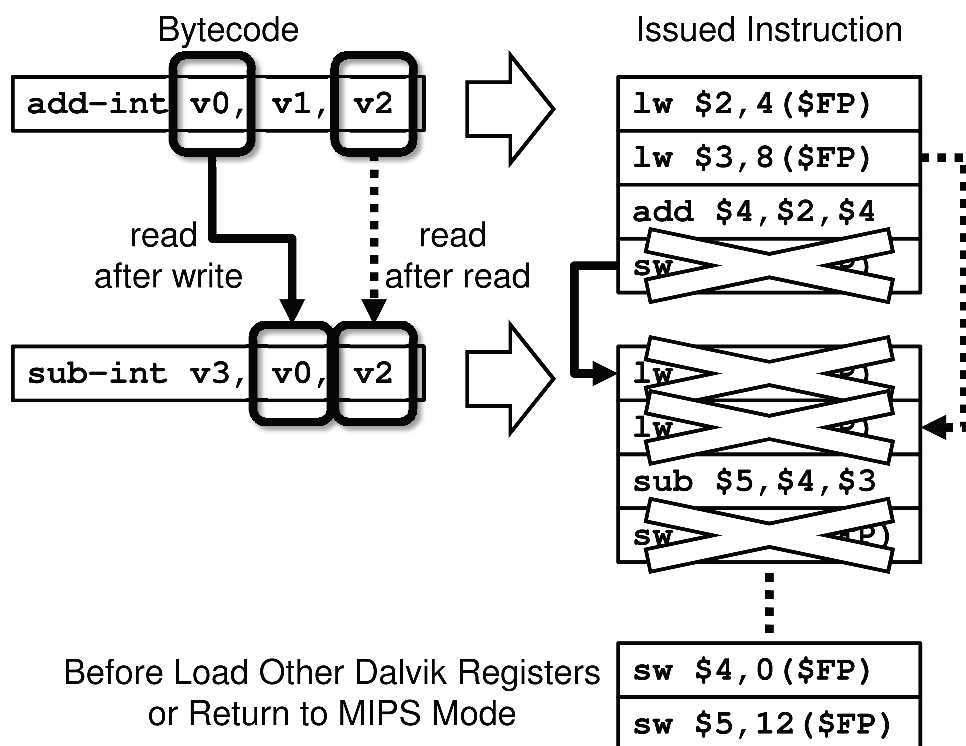

図 6.2: 理想的なバイトコードからデコードされた MIPS 命令列

6.2.1　Dalvik レジスタを格納する専用のハードウェア・レジスタ

考えられる 1 つ目の例は，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・レジスタを設ける方法である．しかし，4.5 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．よって，レジスタファイルの大きさは $32[bit] \times 65,636 = 256[KB]$ となる．現在のスマートフォンをはじめとする，高性能組み込み機器向けプロセッサにおける，L2 キャッシュに迫る大きさとなる．このような巨大なレジスタ・ファイルを，プロセッサに設けるのは現実的でない．よって，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．

6.2.2　Dalvik レジスタの物理レジスタへの静的マッピング

考えられる 2 つ目の例は，Dalvik レジスタの物理レジスタへのマッピングを，静的に行うことである．すなわち，表 5.1 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）に関してはその都度ロード/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都度ロード/ストアする必要がなくなる．6.4 で後述す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，4.5 で述べたように，Android の標準アプリケーションにおいて，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，割り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．また，生成された Dalvik バイトコードにおいて，常に先頭から数個分，静的マッピングの対象となった 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．

6.3　物理レジスタと Dalvik レジスタのマップ

これまで挙げてきた Dalvik レジスタの物理レジスタへのマップ手法は，いずれも課題がある．そこで，より効率の高い物理レジスタにロードされている Dalvik レジスタの情報の記憶を行う，**Dalvik Register Map Table (DRMT)** を提案する．DRMT は命令ジェネレータに接続される．命令ジェネレータがバイトコードより MIPS 命令を発行する際，Dalvik レジスタのマップ状況を DRMT を用いて確認，ロード・ストアを削減できるか判定する．

図 6.3 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 ビット幅，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 個の任意な 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り当て中の 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有効となる．

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．

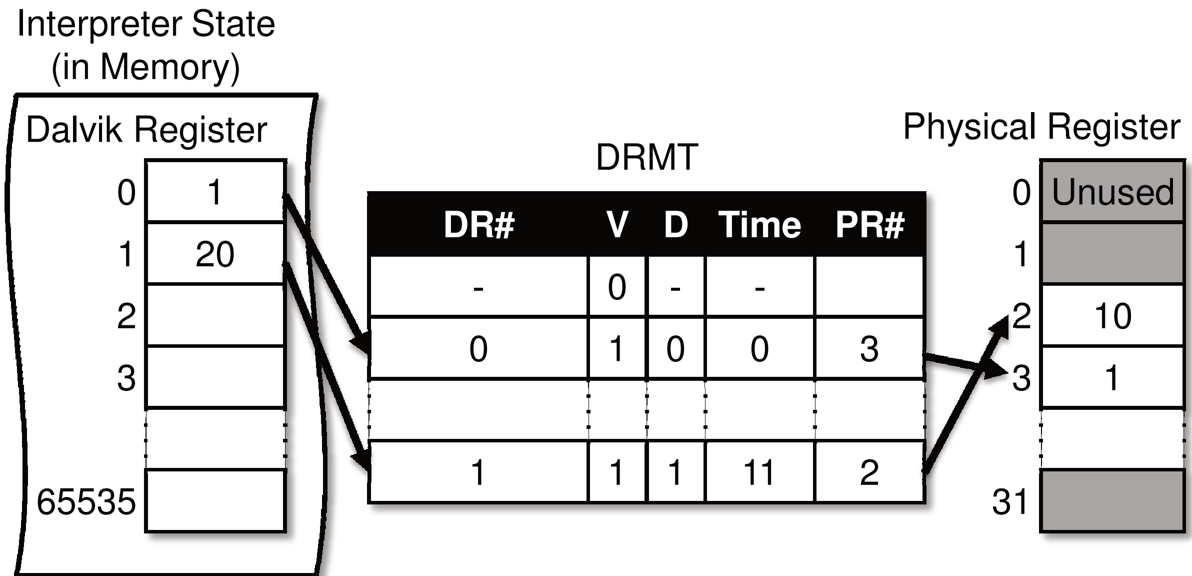

図 6.3: DRMT (Dalvik Register Map Table)

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.

6.4 DRMT の動作

本節では，図 6.4 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 命令の変換結果から出力された，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 レジスタの値間で，加算を行う．加算後，演算結果が含まれる物理レジスタから， 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 レジスタに対応する物理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.

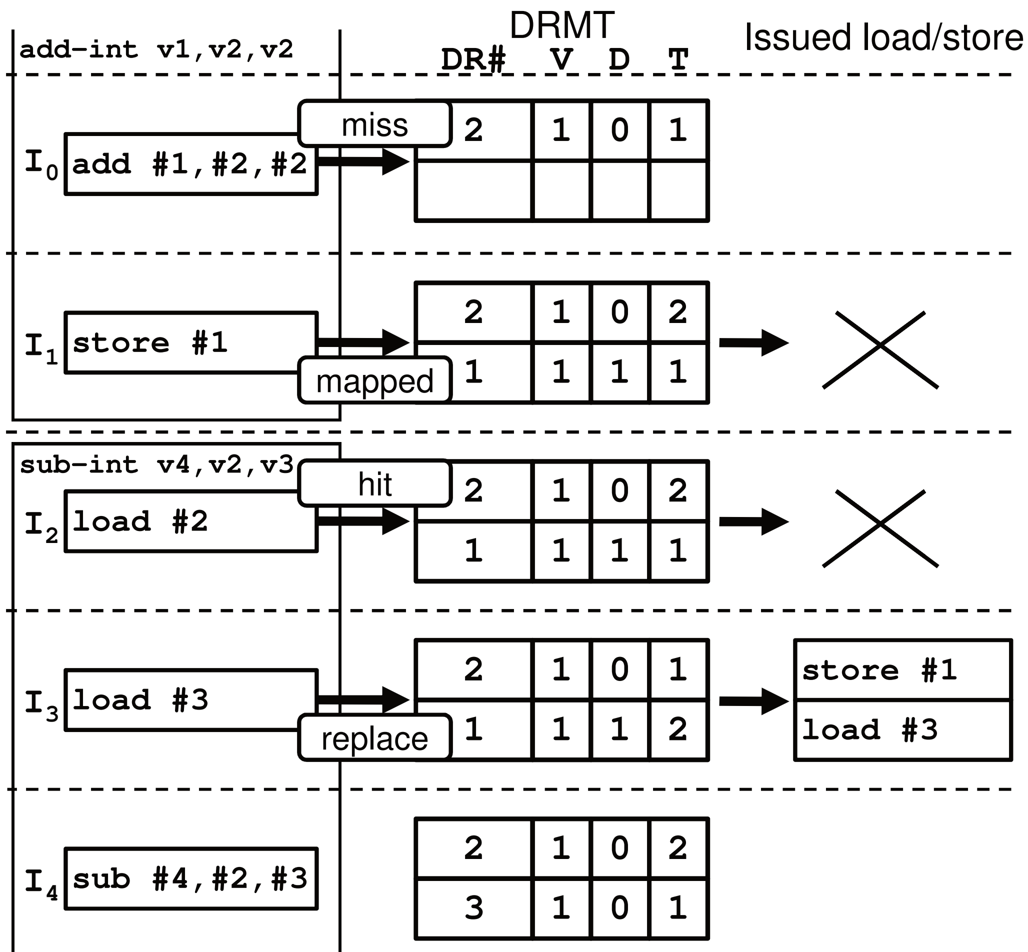

図 6.4: DRMT の動作例

I_1: ストア命令の省略　I_1 は add-int 命令から変換された最後の命令で，ここでは演算結果を#1 に相当するアドレスへストアを行う動作である．しかし，Dalvik レジスタへのストア動作は，DRMT へのヒット・ミスに問わず，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 I_0 で 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 命令から変換された 1 つ目の命令で，オペランドで指された 1 つ目の 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 がエントリされているか探す.

I_1 の発行判定後の DRMT より，#2 は DRMT にマップ，すでに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 命令から変換された 2 つ目の命令である．オペランドで指された 2 つ目の 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必要があるため，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．

6.4.1 命令畳込みとの相違点

3.3.2 にて示したように，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，例えば，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．

[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．その上で，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 [47].

6.5 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，例外情報をレジスタ $EHND 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時に（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時にストアするのが最も効率がよい．

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時，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）を呼び出した時，一旦寿命が尽きることになる．そのような場合には，例えネイティブ・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，一旦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[2].

6.6 ハードウェア上での実装方法

DRMT は静的マッピングに比べて，ハードウェア量の増加は避けられない．しかしその増加は Dalvik アクセラレータや実装先のプロセッサそのものに比べれば軽微である．

DRMT の 1 エントリが保有する情報量は,

Dalvik レジスタ番号 16 ビット

Valid ビット 1 ビット

Dirty ビット 1 ビット

タイムスタンプ DRMT のエントリ数を格納可能なビット幅

物理レジスタ番号 プロセッサのレジスタ番号を格納可能なビット幅

となる．表 6.1 に例として，MIPS，ARM，2 つのアーキテクチャに対して DRMT を適用する場合の DRMT の大きさを．DRMT エントリ数は，MIPS 12 エントリ，ARM 4 エントリとする．これはもともとのレジスタ本数ならびに アクセラレータによって使用されるレジスタを引き，Dalvik レジスタを物理レジスタに保持できる個数を割り振っている．DRMT のエントリ自体は，MIPS アーキテクチャで 324 ビット，ARM アーキテクチャで 92 ビットと，小容量のメモリで済む.

表 6.1: MIPS，ARM 各アーキテクチャにおける DRMT のビットサイズ

種別	MIPS	ARM
Dalvik レジスタ番号	16	16
Valid ビット	1	1
Dirty ビット	1	1
タイムスタンプ	4	2
物理レジスタ番号	5	4
ビット幅	27	23
エントリ数	12	4
総サイズ	324	92

[2]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.

次に，DRMT にアクセスする方法として，各エントリとアクセスしようとする Dalvik レジスタの番号間で CAM を組み，TLB やキャッシュのように一致回路を加える必要がある．16 ビット幅，12 ないしは 4 エントリで一致回路を組むため，回路規模は大きく見えるが，TLB やキャッシュ等他の CAM を用いた機構に比べ規模は小さい．よって，DRMT による回路面積の増加はほかのプロセッサを構成する機構に比べて，大きくない．

第7章　遷移コストの削減機構

本章ではDalvik アクセラレータの実行モードと，プロセッサ・ネイティブな命令の実行モードの上で動作する Dalvik VM との間での，モード遷移コストの大きさと，その改善手法について示す.

まず本課題の背景である，Dalvik アクセラレータの実行モードの遷移コストの大きさと，Dalvik アクセラレータによる性能向上への影響の大きさを示す．続いて 2 つのモード間の遷移コストの削減方法として，2 種類の削減手法を示す.

7.1　実行モード遷移時のコストの大きさ

Dalvik アクセラレータは，実行ステージ以後のプロセッサ資源を併用する．同様に，レジスタやプログラムカウンタについて，既存のプロセッサ資源を併用する．よって，Dalvik バイトコードを実行する前には，Dalvik VM の内部情報へアクセスするためのポインタや，Dalvik レジスタをロード，演算するためのワーク領域となるレジスタの準備が必要である．Dalvik バイトコードのアクセラレーションを中断し，VM へ処理を明け渡すにしても同様で，

こうした処理が遷移のオーバヘッドとして実行時間に加算されるため，遷移の回数が多いと，実行がかえって低速になるおそれがある.

そのオーバヘッドがどの程度のものか確認するため，その遷移にかかる命令数と，遷移の起こる頻度を調査した．まず，MIPS アーキテクチャの Dalvik VM のインタプリタと，Dalvik アクセラレータとの間でこのレジスタの保存，復帰，設定に必要な命令数を計算した．遷移にかかるオーバヘッドは，命令数にして 37 命令であった．このうちのボトルネックを占めるのは，レジスタ s0~ 7 および戻りアドレス ra，スタックポインタ sp の保存と復帰である.

遷移の起こる頻度は，プログラムの処理内容によって大きく変わる．この調査には，Embedded Caffeine Mark を逆アセンブルして得た Dalvik バイトコード群を用いた．ベンチマーク部分は演算が中心であるため，ほぼすべてのバイトコードを高速化可能である．しかし，それ以外の部分では文字列の表示やメソッドの呼び出し，オブジェクトの生成を中心に行われるため，高速化可能な部分がほとんど存在しない場合が多い．このように，高速化可能な命令が連続する部分は限られている.

従来の Dalvik アクセラレータ設計，提案では，バイトコードごとに遷移の可否を選択してきた．アクセラレータでの実行不可能な命令に遭遇したとき，Dalvik VM に戻る．Dalvik VM で実行すべきバイトコードを 1 つ実行したら，Dalvik アクセラレータに処理を戻して，そこからアクセラレータで実行すべきか可否を判断した．そのため，アクセラレート不可能な命令が連続する場合，上記の遷移コストが常時発生す

ることとなり，効率は大幅に低下する．そこで，高速化可能な命令が連続していないところでは，アクセラレータに遷移させないようにする，という手法により遷移回数を削減する必要がある．

Embeded Caffeine Mark の Matrix を用いて，翻訳後に実行される MIPS 命令数を計算した．すべてのバイトコードを VM のみで実行した場合，のべ 4422 命令が実行される．対して，アクセラレータで実行可能なバイトコードを Dalvik アクセラレータの変換対応表で翻訳すると，のべ 2361 命令となる．しかしモード遷移は 51 回発生するため，オーバヘッドが $51 \times 37 = 1887$ 命令分加わり，のべ 4248 命令となる．このように，アクセラレータを併用しても毎回モード遷移していては，Dalvik VM 単体実行時に比べ性能向上に結びつかない．

7.2 実行モード遷移コストの削減

事前評価の結果より，モード遷移時のコストを削減することが，Dalvik アクセラレータの性能向上につながることが判る．本来のアクセラレータの効果を失わせないためには，適用が必要である．

以上の評価結果から，モード遷移時のコストを削減できれば，Dalvik アクセラレータの性能向上をより発揮させることができる．

そのために，以下に示す 2 種類の方式を提案する．1 つは実行しようとする Dalvik バイトコードがハードウェア実行できる，もしくは，できない命令が連続しているか予測する．そして連続していないならば遷移を回避する手法である．

2 つ目の手法はレジスタの保存，復帰処理を高速にするために，Dalvik VM と Dalvik アクセラレータが重複して使用するレジスタについて，もう 1 セット設ける．これをレジスタウィンドウとし，高速にレジスタセットを切り替える．

7.2.1 Dalvik アクセラレータのモード遷移予測

図 7.1 にアクセラレータによる実行のフローチャートを示す．Dalvik VM の初期化の後，Dalvik VM から Dalvik アクセラレータへ処理が移りバイトコードが実行される．Dalvik アクセラレータで実行不可能なバイトコードをフェッチすると，代わりに Dalvik VM がそれを実行するよう，制御を移す．その後，通常は再び Dalvik アクセラレータへ制御を戻す.

大きなコストがかかるのは各モードの遷移である．モード遷移の度に各モードの準備をするためのオーバヘッドが発生する．そこで，次の Dalvik バイトコードも Dalvik アクセラレータで実行できないと判断したら，モード遷移を抑止し，VM での実行を続ける．こうして遷移コストを削減し，Dalvik アクセラレータの効果を最大限に生かせるタイミングを実現する．

Dalvik バイトコードで記述された Android アプリケーションは，プログラムの自己書き換えがない．よって，分岐する場合を除いて，次に実行すべき Dalvik バイトコードがアクセラレート可能かどうか判断できる．

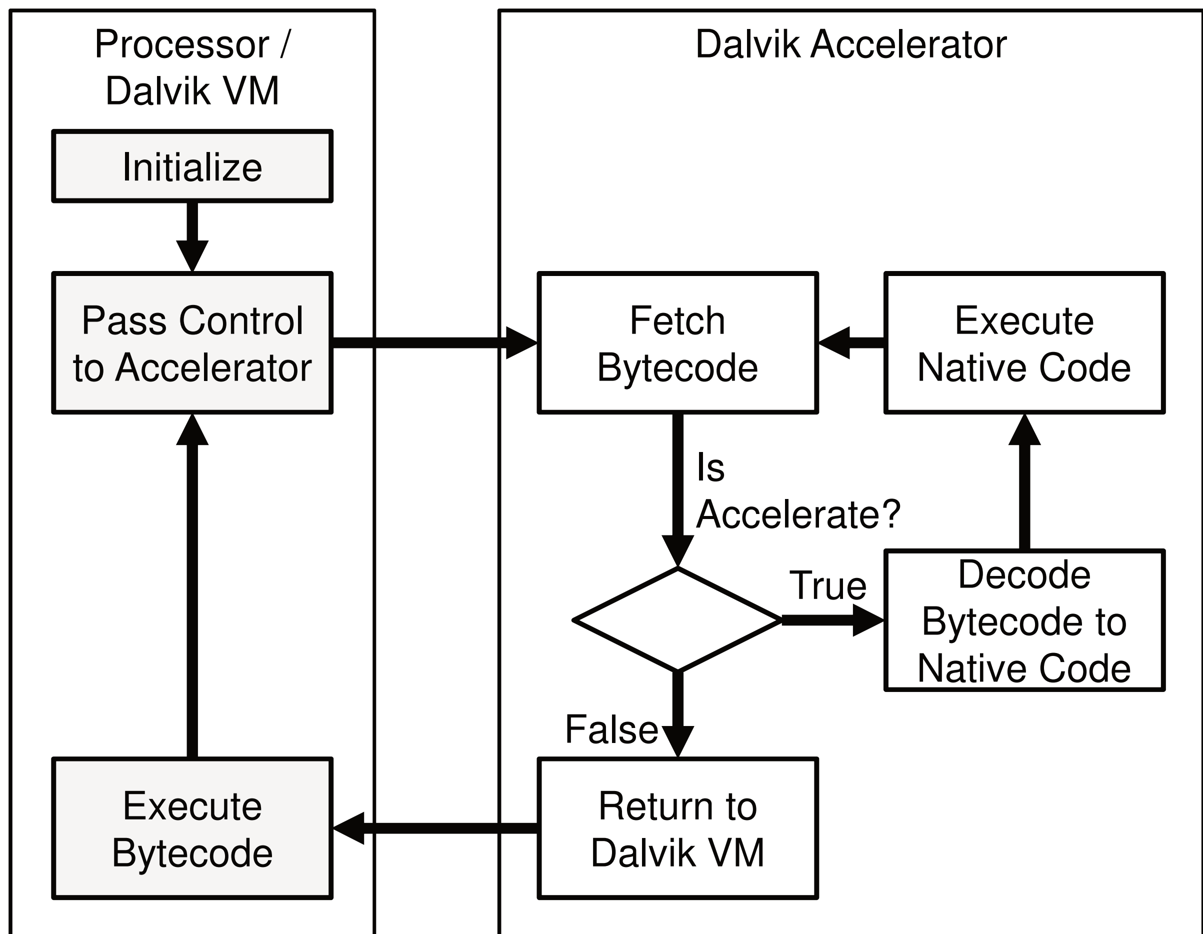

図 7.1: Dalvik アクセラレータの実行フローチャート

Dalvik VM が Dalvik アクセラレータにモードを切り替えるか判断するのは，Dalvik バイトコードのフェッチ・デコード後である．Dalvik VM が Dalvik バイトコードの命令フィールドを参照して，その Dalvik バイトコードがアクセラレート可能かどうかの判断をする．これにより，アクセラレート不可能な Dalvik バイトコードが連続する場合であっても，モード遷移は最初の 1 回だけで済む．このように，Dalvik VM にてアクセラレートの判断を行い，Dalvik アクセラレータへの切り替えを削減することで，余分なモード遷移を減らすことができる．これらの改良した遷移フローチャートを図 7.2 に示す．

Embedded Caffeine Mark の Matrix の場合，高速化不可能な Dalvik バイトコードが連続している箇所がいくつか見られる．そのうちには，最大 18 個連続している箇所も存在する．こうした箇所では，実行モード遷移は最初の 1 回だけ行い，後続する Dalvik バイトコードは Dalvik VM で処理すればよい．この結果，実行モード遷移回数は 51 回から 10 回に削減した．

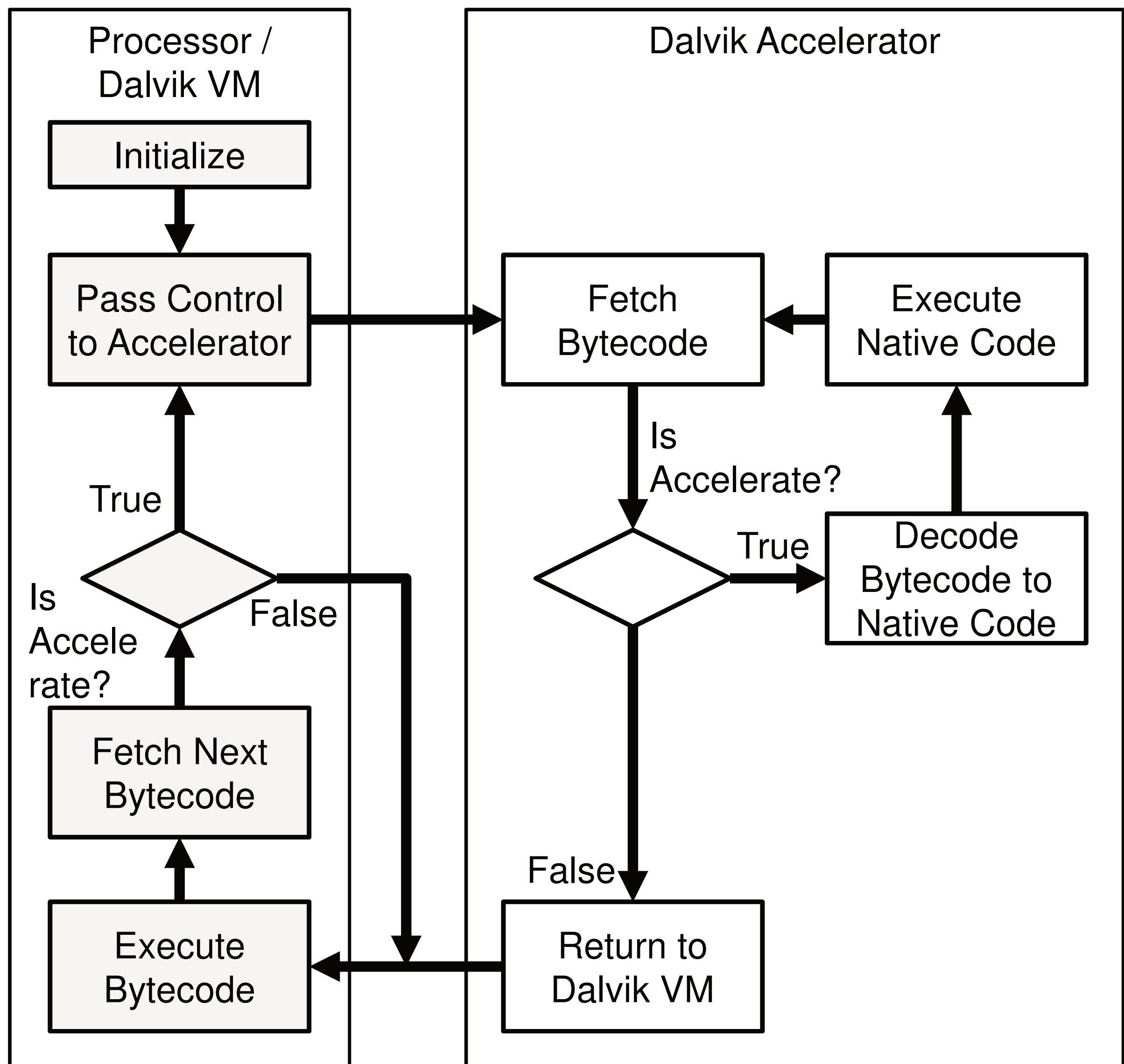

図 7.2: 実行モード遷移を削減する Dalvik アクセラレータの実行フローチャート

7.2.2　2 つの実行モードに対応するレジスタウィンドウの設置

さらに実行モード遷移 1 回あたりのオーバヘッドを減らすために，レジスタを追加する．Dalvik VM と Dalvik アクセラレータとで共通した目的に使われる物理レジスタは，値を保持しておいてよい．一方それ以外の物理レジスタは逐一上書きが起こるので，値の破壊を防ぐためには，実行モード遷移の際に退避と復帰をさせる必要がある．

Dalvik アクセラレータの実装先例として，MIPS アーキテクチャの場合 32 本の物理レジスタのうち，8 本は表 7.1 のような目的で，Dalvik VM と Dalvik アクセラレータ間で共有する．それ以外の 24 本の物理レジスタについて，退避と復帰を行わなければならない．

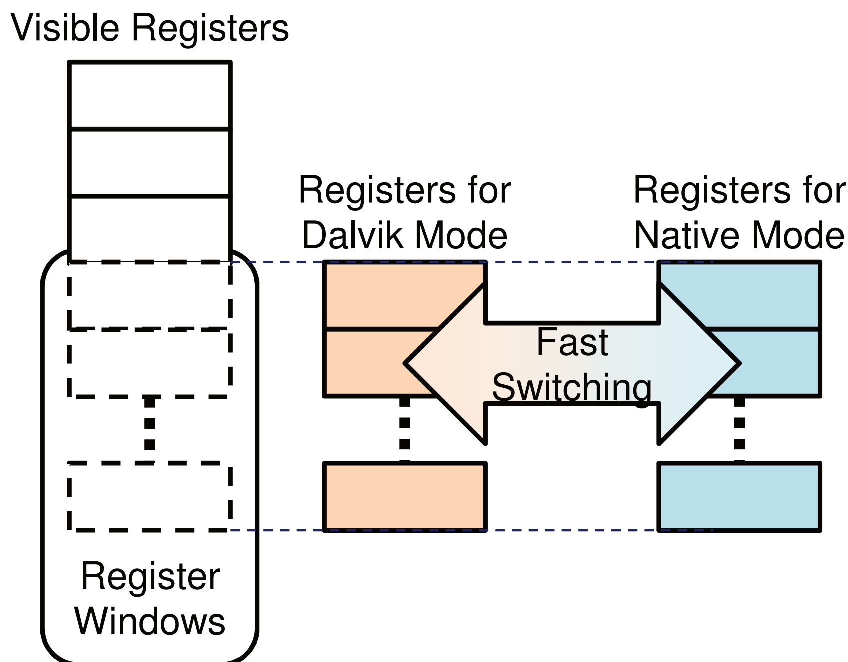

図 7.3: レジスタウィンドウのアーキテクチャ

この共有されない 24 本のレジスタについて，レジスタウィンドウを新たに搭載する．すると，32 本のレジスタの内訳が図 7.3 のようになる．8 本は Visible Registers に位置し，前述の固定された用途に使われる．24 本は Register Windows に位置し，新たに 24 本ずつ用意された Dalvik モード用のレジスタとネイティブモード用のレジスタをそれぞれ参照できる．これによりレジスタの退避と復帰を行うことなく，参照するレジスタを切り替えるだけで高速なモード遷移が可能となる．よってオーバヘッドとなっていたレジスタの退避と復帰を行うための命令が削減される．レジスタの退避と復帰はオーバヘッドの 37 命令中 20 命令を占めていたため，1 回の遷移にかかる命令数は 17 命令に減ることになる．

表 7.1: 8 つのレジスタの目的

レジスタ	目的
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
IBASE (r19)	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
EHND (r20)	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の参照
ESTAT (r22)	発生した例外の種類が格納されるレジスタ
EPC (r23)	例外を起こしたバイトコードのアドレス

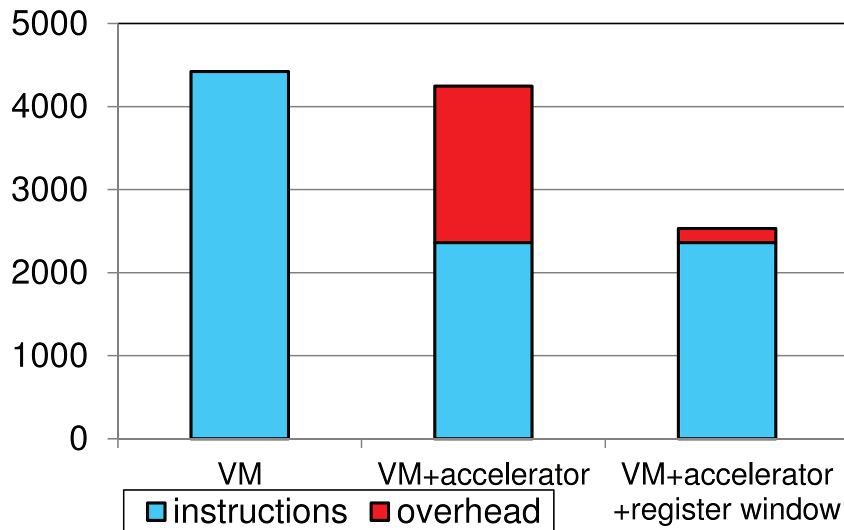

図 7.4: 実行命令数とオーバヘッドの削減量

レジスタウィンドウの切り替えのために，専用命令を定義し，Dalvik アクセラレータのターゲット・プロセッサへ追加する．Dalvik VM から Dalvik アクセラレータへのモード遷移では Dalvik VM のインタプリタから，Dalvik アクセラレータから Dalvik VM への遷移では Dalvik アクセラレータのデコーダから，この命令を発行する．この命令が実行されたとき，参照されるレジスタウィンドウを一括で切り替える．切り替えた先のレジスタウィンドウには以前利用していた値がそのまま格納されているので，何の操作もなく再び利用することができる．

7.2.3 性能評価

この 2 つの手法を組合わせることで，実行モード遷移回数が削減され，さらに実行モード遷移 1 回あたりのオーバヘッドも半減する．Embedded Caffeine Mark の Matrix の場合，図 7.4 に示すように発行される MIPS 命令数はのべ 2,531 命令まで削減される．これは VM のみの実行に比べ，1.7 倍高速化されたことになる．

7.3 まとめ

Dalvik アクセラレータでは，実行不可能なバイトコードを扱う際，Dalvik VM で実行するため実行モード遷移が必要である．それに当たり，Dalvik アクセラレータと Dalvik VM で使用目的の異なるレジスタの退避，復帰が必要であった．この実行モード遷移の実行コストは無視することができないほどに大きく，実行モード遷移回数が多いほど膨れ上がる．

そこで，アクセラレート可能な命令，不可能な命令の連続する箇所での実行モード遷移の可否を予測する．これにより，アクセラレート不可能な命令を実行後，常に Dalvik アクセラレータに制御を移さず，実

行モード遷移のコストを下げる手法を提案した．また，レジスタウィンドウの追加により，実行モード遷移の度に行われた物理レジスタの退避，復帰が不要とする手法を提案した．この 2 つの手法を組合わせることで，Dalvik アクセラレータの本来持つ高速化効果を充分に生かすことができる．

第8章　プロセッサシミュレータ上でのAndroidの実行

本章では，プロセッサ・シミュレータ SimMips[61] に対して，Dalvik アクセラレータを搭載しその効果を検証するために必要な改造を挙げる．そして改造の結果，SimMips が現在の携帯情報端末向けプロセッサに近い性能の傾向を示すか，Dalvik VM 上でベンチマークプログラムを実行し測定する．その結果から，より正確に Dalvik アクセラレータの性能向上が確認できるプロセッサ・シミュレータの構成となったか評価する．

8.1　SimMips の概要

SimMips は東京工業大学 吉瀬研究室が開発した，教育用コンピュータ MieruPC [63] 向けの MIPS ライクなアーキテクチャを持つプロセッサ・シミュレータである．UNIX ベースの環境で動作する．

SimMips は App モード，OS モード，MieruPC モードと，3 つのプロセッサ実行モードを有する．App モードは，uClinux ベースのクロスコンパイル環境下で作成した，単体のプログラムのシミュレーションに対応するモードである．OS モードは，例外，TLB を利用する OS レベルのシミュレーションに対応するモードである．MieruPC モードは，App モードから更に，キーボード入力，MieruPC 本体に接続される液晶ディスプレイの出力シミュレーションを行い，MieruPC の動作全般をシミュレーションするモードである．

このうち，OS モードにおいて，CPU エミュレータ QEMU 向けに配布されている，Linux 環境のディスクイメージを流用することで，SimMips 上で MIPS アーキテクチャ向けにコンパイルされた Linux を作動できることが確認されている．

Dalvik アクセラレータの実装において，SimMips をターゲットとした理由を以下に挙げる．

Linux が作動すること　Android は基盤となるオペレーティングシステムとして，Linux カーネルを採用している．よって，Linux が実行可能な環境でなければならない．また，Android を動作させる Linux カーネルには，プロセス間通信を担当する binder，Android 独自のログシステムなど，独自のデバイスドライバを組み込む必要がある．

SimMips は前述のとおり，すでに Linux が動作可能であることが確認されている．よってこの条件を満たす．

変更可能であること　本研究では，既存のプロセッサに対して Dalvik アクセラレータを搭載し，その結果向上した性能を評価することが目標にある．よって，プロセッサを構成する，ソフトウェアのソースコード，ハードウェア記述言語（HDL）が公開されていなければならない．また Dalvik アクセラレータに相当するコードを追加できるよう，変更が可能でなければならない．

SimMips は C++ で記述され，GPL ライセンスの下，ソースコードが公開されている．よってこの条件を満たす．

Android がサポートするアーキテクチャであること　1.2 で述べたように，Android は現在，ARM，x86，MIPS などのアーキテクチャで稼働する．よって，アクセラレータを搭載する先のプロセッサ・アーキテクチャは，Android の稼働するアーキテクチャでなければならない．

SimMips は MIPS シミュレータであることから，この条件を満たす．

アクセラレータを実装する方法は，シミュレータや HDL で記述された実装など，複数の適用方法が考えられる．しかし，上記の条件を満たすシミュレータや HDL による実装は，知る限りの範囲では SimMips に限られた．

一方，SimMips には，Dalvik アクセラレータを実装し正確な評価を行うにあたって，その精度に不十分な個所が存在する．その個所に対して変更を加え，現在の携帯情報端末向けプロセッサに近い構成の元，サイクル・アキュレート且つオペレーティングシステムが稼働可能な，プロセッサ・シミュレータとした．

8.2　SimMips のアーキテクチャ

ここでは，SimMips の詳細な内部設計について示す．

8.2.1　命令の実行

SimMips の命令セットは MIPS32 Revision 1 ライクである．プロセッサ内部は，7 ステージ（IF: 命令フェッチ – ID: 命令デコード – RF: レジスタフェッチ – EX: 実行 – MS: メモリストア – MR: メモリリード – WB: レジスタライトバック）のパイプラインを持つ．そして，各ステージを模した，7 つの関数を順次実行することで，シミュレーションを実現している．図 8.1 に各ステージを模した関数と，SimMips が持つプロセッサ内部情報との関係を示す．プロセッサ内部情報として，プログラム・カウンタ（pc），フェッチした命令列（binary），オペコード（opcode）といった変数を有する．各関数は，それぞれのステージに対応する動作を振る舞い，これらの変数を更新することでシミュレーションを行う．

SimMips の実行モードは，MIPS 命令を 1 サイクルでシミュレーションするシングルサイクル・モードと，1 サイクルごとにプロセッサ内の各ステージを順次実行し，5~7 サイクルかけてシミュレーションする，マルチサイクル・モード，2 つの実行モデルを持つ．図 8.2 にサイクル当たりのシミュレーションするステージ数の相違を示す．シングルサイクル・モードでは，各ステージを模した関数を 1 サイクルのシミュ

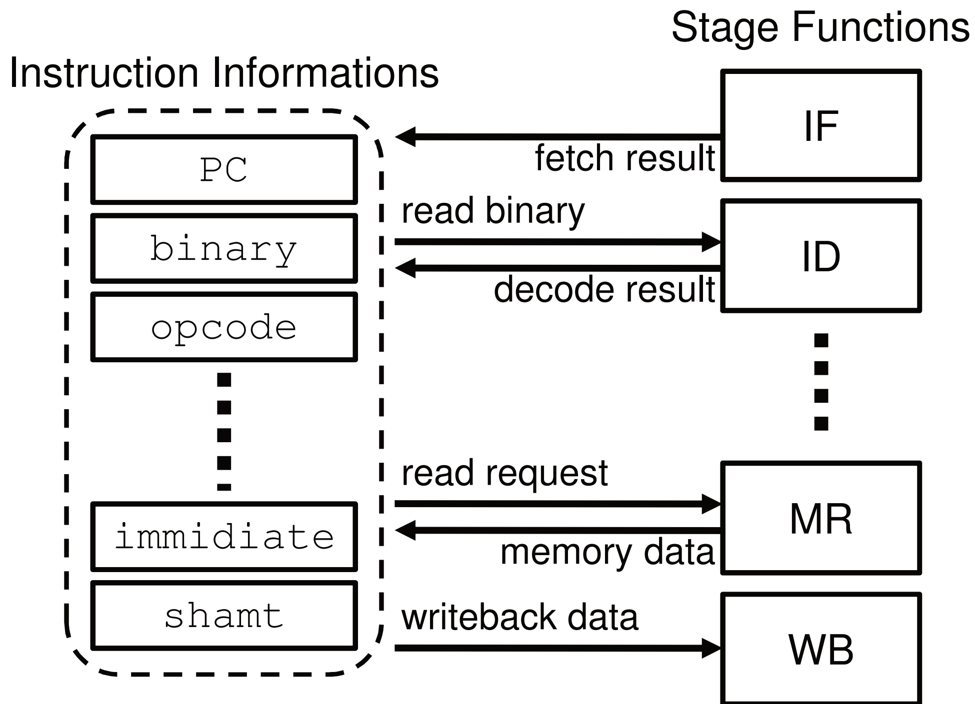

図 8.1: SimMips のプロセッサ情報と各ステージの関係

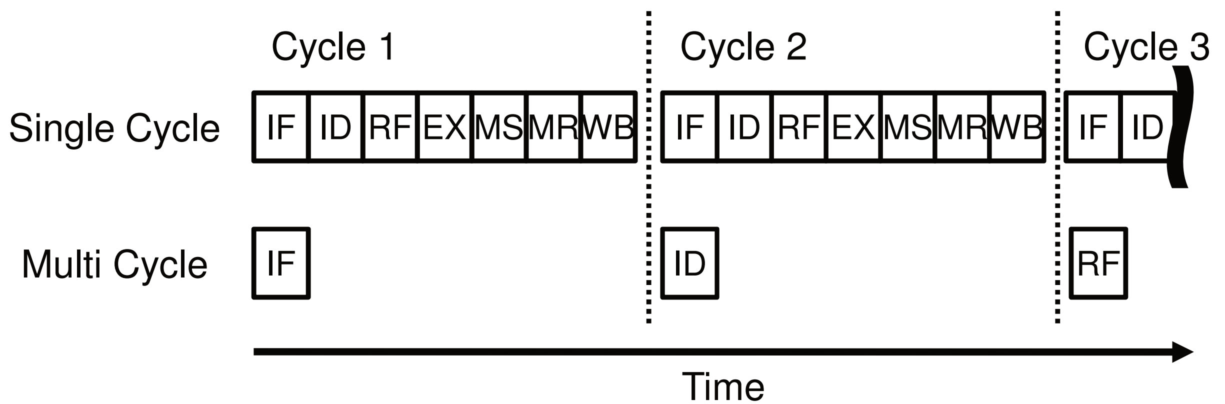

図 8.2: シングルサイクルモードとマルチサイクルモードのシミュレーション動作

レーション中に一通り呼び出す．一方マルチサイクル・モードでは，1 サイクルのシミュレーションにつき，命令の実行に必要なステージを模した関数 1 つずつを呼び出す．

8.2.2　メモリアクセス

SimMips におけるメモリ・アクセスについて，シングルサイクル・モードでは 1 サイクル中の処理に収まっている．マルチサイクル・モードでは IF(フェッチ)，MS(メモリ・ストア)，MR(メモリ・リード)，各

ステージでのメモリアクセスは 2 サイクルでメインメモリへアクセスすることを想定しており，レイテンシは変更できない．

キャッシュメモリは搭載されていなく，すべてのメモリアクセスは等しいレイテンシで行われる．

8.2.3 周辺機能

周辺機能としてコプロセッサ CP0 が実装されており，例外処理のハンドリング，TLB の管理を行う．このほか，シリアル I/O のコントローラを搭載する．シリアル I/O コントローラは，キーボードからのデータ入力と，標準出力へのデータ出力をサポートする．オペレーティングシステムを稼働させる場合は，コンソール入出力を，このシリアル I/O に設定する．Linux ではカーネルパラメータに `console=ttyS0` と指定し，シリアルコンソールの有効ならびに既定のシリアルポートをコンソールに設定する．

補助記憶装置のシミュレーションは実装されていない．一方，指定したファイルを，物理アドレス空間の特定の範囲にマップする機能を有する．これを利用し，SimMips 上で Android を実行する場合，起動させる Linux の root パーティションとなるディスクイメージを，SimMips の物理アドレス空間にマップしている．そして，カーネルパラメータで，ルートパーティションの存在する物理アドレス空間を指示することで，擬似的にストレージを実現している．

8.3 SimMips の変更点

本項では，SimMips へ行った変更点について示す．これらの変更点を加えたのち，サイクル・アキュレート且つオペレーティングシステムが動作可能な，プロセッサ・シミュレータとした．

8.3.1 パイプライン化

既存のシングルサイクル実行ならびにマルチサイクル実行に加えて，新たに実行モデルを追加した．これまでの両実行モデルは，現在のプロセッサの実装とはかけ離れている．アクセラレータを実装し，性能評価を行うにあたっては現実的なモデルではない．よって，SimMips に対してパイプライン実行モデルを追加することで，サイクル・アキュレートなシミュレーションモデルとする．

SimMips は前述のとおり，各ステージを模した関数が順次実行されることでプロセッサのシミュレーションを行う．このとき，フェッチした MIPS 命令列からメモリもしくはレジスタに書き込まれるデータ列まで，命令実行に必要なプロセッサ内部情報をシミュレータ上で 1 つしか有していない．

パイプライン実行モデルでは，各ステージ間毎にプロセッサ内部情報を有する，構造体配列 `pipeline[7]` を設けた．構造体 `pipeline_t` の内部は，シングルサイクル実行，マルチサイクル実行における，プロセッサ内部情報と同等である．図 8.3 に，パイプライン実行モデルにおける，各ステージを模した関数と構造体配列の関係を示す．これにより，各ステージがクロック単位で並列に動作するシミュレーションを行えるようにする．

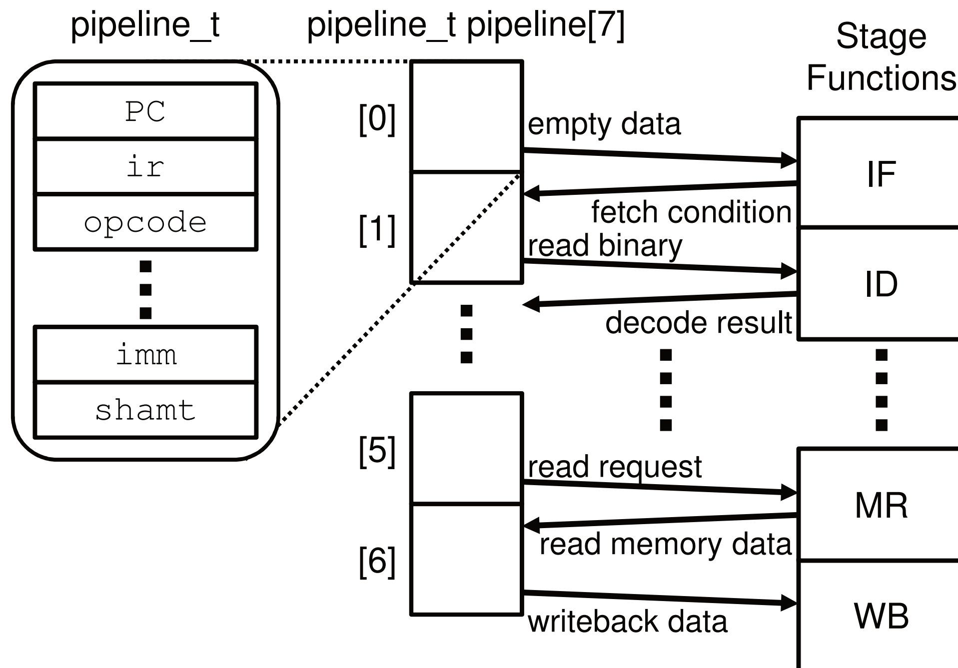

図 8.3: SimMips におけるパイプライン実行

サイクル毎のシミュレーションは，パイプラインの末端，ライトバック・ステージよりシミュレーションを行う．ストールなどパイプラインを停止させる要因がなく，次ステージへプロセッサ内部情報を受け渡せるのであれば，後続するステージのシミュレーションを継続する構造となっている．

パイプライン実行では，このほかに追加した動作がある．分岐命令の成立や例外発生といった，PC が更新される動作が発生したとき，パイプラインをフラッシュする動作を追加した．また，分岐命令については，後続するアドレスにある命令を投機的にパイプラインに続けて投入しており，常時不成立側に静的分岐予測する動作である．

8.3.2　メモリアクセスに伴うレイテンシとキャッシュのシミュレーション

既存の SimMips のメモリアクセスは，レイテンシがほぼ皆無であり，キャッシュのシミュレーションが行われていない構成である．これは現在のコンピュータのメモリアーキテクチャにおいては非現実的である．正確なシミュレーションのため，キャッシュの実装，レイテンシの発生とそれに伴う命令パイプラインのストールを実装する．

キャッシュの構成，容量は，現在の携帯情報端末向けプロセッサの構成に近づける．2 レベルのキャッシュであり，L1 キャッシュは命令，データそれぞれ 32KB，L2 キャッシュは 256KB とした．実装するキャッシュはデータをプロセッサ–L1 キャッシュ–L2 キャッシュ–メインメモリ間で転送しているのではなく，アドレスのみ保持し，各レベルのキャッシュに存在するか記憶している．

メモリアクセスが発生すると，各レベルのキャッシュに指示されたアドレスがエントリされているか確認

する．その結果から，レイテンシをシミュレーションするためのレイテンシ・カウンタに加算される．毎サイクルの実行毎に，レイテンシ・カウンタが 0 でなければ，デクリメントする．その間，命令パイプラインをストールさせることで，メモリアクセスによるレイテンシをシミュレーションする．

各階層へのレイテンシは，L1 キャッシュがともに 1 サイクル，L2 キャッシュが 8 サイクル，メインメモリが 100 サイクルとした．

8.3.3　未定義命令における例外発行

SimMips では，MIPS32 Revision 1 ライクな命令セットを実装している．未定義の命令については，未定義命令エラーを発生し，SimMips はシミュレーションを中断する．

しかし，MIPS アーキテクチャの Linux や，Android で用いられる標準 C ライブラリでは，アセンブリ言語やインラインアセンブラにより，上位リビジョンである MIPS32 Revision 2 の命令がコードに含まれる．よって，Android のソースコードで make を実行する際，明示的にターゲット・アーキテクチャを MIPS32 Revision 1 に指定しても，同アーキテクチャでは実行できないバイナリが生成される．

通常の CP0 が搭載された MIPS32 プロセッサ上では，このような命令に対して，未定義命令例外が発生する．そして例外によって，オペレーティングシステム内のハンドラに処理が移り，ソフトウェアによる代替実行が行われる設計となっている．

SimMips でも同様に未定義命令が入力されたとき，未定義命令例外が発生するよう動作を変更した．所定の例外ハンドラ・アドレスへジャンプし，例外の要因として未定義命令であることがわかるよう SimMips へ変更を加えた．

これらの実装を行った結果，パイプライン・プロセッサのシミュレーションを有効にした SimMips 上で Linux を起動し，その上で Dalvik VM の実行が可能となった．

8.4　Android の起動環境の構築

SimMips は補助記憶装置のシミュレーションや外部ハードウェア，フレームバッファに対応していない．よって，AOSP より入手したソースコードより，ビルドした実行イメージのままでは，起動することやコマンドの実行はできない．そして，Linux カーネルについても，Android 向けの特殊なカーネルモジュール，コンフィギュレーションがあるため，SimMips ですでに動作が確認されている QEMU 向け Linux カーネルをそのまま流用できない．

8.4.1　カーネルの設定

Linux カーネルのコンパイルについて，MIPS アーキテクチャ向けのコンフィギュレーションの中から，MIPSSim コンフィギュレーション（mipssim_defconfig）[1]を選択，そこから Android に必要なカーネル・モ

[1]MIPS Technologies による MIPS シミュレータ向けの設定．MIPSSim 自体は プロプライエタリなソフトウェアである．

ジュールをカーネルに組み込む選択した．

Linux カーネル・コンフィギュレーションにて，主に有効，変更した項目は次のとおりである．

General setup → Enable the Anonymous Shared Memory Subsystem Android の共有メモリシステム ashmem を有効にする．

Device Drivers → Misc devices → Staging drivers → Android → Binder IPC Driver Android のプロセス間通信ドライバ．

Device Drivers → Misc devices → Staging drivers → Android → Android log driver Android 独自のログシステム．評価を行う際に必要である．

Kernel hacking → Default kernel command string カーネルパラメータのデフォルト引数．値を `console=ttyS0 rd_start=0x81000000 rd_size=67108864 ramdisk_size=65536 init=/init` に変更した．

カーネルパラメータの各要素は次の意味となる．シミュレーション時に物理アドレス空間に配置する，root パーティションとなるディスクイメージに関する設定が占める．

console コンソール出力先の指定．標準入出力とつながった，SimMips のシリアル I/O へ接続するので，シリアルポート ttyS0 を指定する．

rd_start RAM ディスクイメージの開始番地を指定する．

rd_size RAM ディスクイメージのサイズを指定する．

ramdisk_size RAM ディスクのサイズを指定する．カーネル設定で変更しないかぎり規定値は 16384KB である．より大きなディスクイメージを RAM ディスクに展開するため，サイズを指定する．

8.4.2 シェルへのアクセス

実行イメージについて，Android 独自の init が読み出す init スクリプト`/init.rc` の変更により，Android の多くのサービスを停止させ，SimMips のシリアルコンソールよりシェルにアクセスできるように変更を加えた．

Android のアプリケーションは，通常 Dalvik VM や共有ライブラリをロードした zygote より，fork システムコールによるプロセス複製を経て起動するが，dalvikvm コマンドを経由して起動することも可能である．Dalvik VM のクラス・ライブラリには，java.io パッケージのクラスも含まれている．よって，標準入出力を用いた Java プログラムも実行できる．

図 8.4 にコンソールから Dalvik VM を起動する例を示す．引数体系は Java における java コマンド同様の実装がある．オプションには，-Xbootclasspath オプションで Android のクラス・ライブラリを指定し，-classpath オプションで実行したいクラスのを含む dex ファイル（もしくはそれをパッケージした jar ファイル）を与える．そして Static 宣言の main[] メソッドを持つ，実行したいクラス名を指定する．

```
# dalvikvm -Xbootclasspath:/system/framework/core.jar -classpath /data/ecm.jar
    CaffeineMarkEmbeddedApp

Sieve score = 558
Loop score = 585
Logic score = 509
String score = 2135
Float score = 250
Method score = 441
Overall score = 582
#
```

図 8.4: コンソールからの Dalvik VM 実行例

8.5 評価

SimMips に加えた変更が，アクセラレータを実装する環境として妥当であるか，ベンチマークソフトを動作し，すでに明らかになっている既存の Android 搭載端末 Google Dev Phone 1 （GDD1）[27] のベンチマーク結果を比べることで評価した．シミュレータと該当端末にて，Dalvik VM 上にてベンチマークプログラムを稼働させ，比較した．

8.5.1 評価環境

ベンチマークプログラムには Pendragon Software の Embedded CaffeinMark (ECM)[52] を用いた．ECM は Java VM 向けベンチマークソフト CaffeineMark から，グラフィック，UI 要素のベンチマークを取り除いた，組み込み機器向けに評価項目を絞ったベンチマーク・ソフトウェアである．6 つのベンチマークと総合結果について，Pentium 133MHz，Windows 95 上で稼働する，JDK 1.1 のインタプリタ VM 環境におけるベンチマーク結果を 100 に正規化したスコアを出力する．スコアが高い程高性能である．

6 つのベンチマークの内容 [52] は以下からなる．また，これらの結果より，総合スコアを Overall として出力する．

Sieve エラストテネスのふるい

Loop ループ処理

Logic Boolean 型変数へのブール代数の代入，ブール演算とその比較

String 文字列処理

Float 浮動小数点演算

Method メソッド呼び出し

表 8.1 に変更を加えた SimMips と GDD1 の主な仕様を示す．SimMips のシミュレーション実行時のパラメータは，アクセラレータを搭載する環境を想定し，現在の携帯情報端末向けプロセッサに近づけた．差異の大きな部分として，命令パイプラインと，分岐予測の動作が挙げられる．

8.5.2 ベンチマーク結果

図 8.5 に改造した SimMips と GDD1 との比較結果を示す．SimMips が GDD1 に対して，Float を除くベンチマークについて，半分以上のスコアを示した．アーキテクチャの違いはあるが，この差は主にプロセッサのパイプラインの構成が，スカラとスーパスカラの違いによるものから生じたとみられる．

比較した GDD1 の搭載する ARM1136EJ-S の命令パイプラインは，演算とロード/ストアの 2 命令同時発行スーパスカラである．一方，改造を加えた SimMips の命令パイプラインはスカラである．このことから，命令パイプラインが実行できる命令のスループットは半分である．そのため性能が半分になっていると考えられる．

ベンチマーク項目の中で，Float について著しく劣る．これは SimMips に浮動小数点命令が実装されていないためである．Android のコンパイル時に，浮動小数点命令の利用の可否に応じて，Dalvik VM の機械語インタプリタの内容が変更される．MIPS アーキテクチャの Android では，浮動小数点命令の利用が無効な場合，Dalvik VM の機械語インタプリタは標準 C ライブラリに組み込まれている，ソフトウェア浮動小数点演算ルーチンを呼び出していることで実現している．

8.6 まとめ

Dalvik アクセラレータを実装し，その性能向上を評価する環境に必要な要件を挙げた．そしてそれに満たすシミュレータもしくは HDL 実装に，SimMips を用いることを示した．

シミュレーション・モデルを，現在の携帯情報端末向けプロセッサに近づけるために，SimMips へ行った改造を示した．そして一般的な Android 端末とは異なるハードウェア構成に合わせた，Android のビルド

表 8.1: 評価環境の主なスペック

	SimMips	Google Dev Phone 1
CPU	MIPS32 Revision 1	ARM1136EJ-S
キャッシュ	L1: 32KB /L2: 256KB	L1: 32KB /L2: 256KB
命令パイプライン	7 ステージ スカラ	8 ステージ 2 命令スーパースカラ
分岐予測	静的（不成立側を投機実行）	動的 (bimode)
プロセッサの動作クロック	500MHz	528MHz
主記憶の容量	192MB	96MB

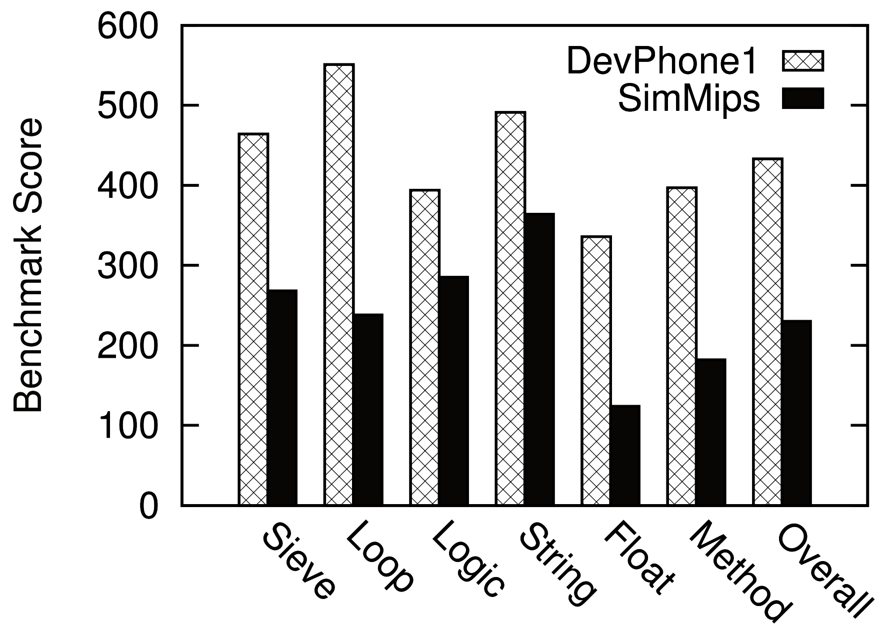

図 8.5: SimMips と Google Dev Phone 1 のベンチマークスコア

イメージ改変について示した．この結果，改造した SimMips のパイプライン実行モードにおいて，Linux を起動させ，更に Dalvik VM を起動できることを確認した．

そして，改造した SimMips が既存の携帯情報端末向けプロセッサと比較して，妥当な性能の傾向が表れているか評価を行った．SimMips のスペックを GDD1 に近づけ，評価を行った結果，およそ半分以上の性能を示した．この性能の差は，両評価の命令パイプラインの構成の違いによるものだと考えられる．よって，SimMips に加えた改造により，Dalvik アクセラレータを実装し，妥当な性能が得られる評価環境であることが確認できた．

第9章　評価

本章では，Dalvik アクセラレータのメモリアクセスを削減する DRMT によるメモリ削減効果について，その評価結果について示す.

評価は 2 つの項目からなる.

1. DRMT によるロード・ストア削減効果の評価．Dalvik アクセラレータを組み込む MIPS アーキテクチャだけでなく，Android アーキテクチャへ実装を想定した評価も含まれる.
2. DRMT により削減された MIPS 命令列と，Dalvik VMJIT が生成するネイティブ・コードの間で生成される命令列を比較．JIT に比べて効率的なネイティブ・コード生成が行われているか比較する.

9.1　DRMT によるロード・ストアの削減効果

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．6 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/ストア命令の生成を回避できる．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/ストアの生成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，常に，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.

9.1.1　評価環境

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 9.1 に示す．デコーダが生成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 個以下である．

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．DRMT の構成を 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と変化させて評価を行った．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．DRMT の構成はフルセットアソシアティブ構成とした．

続いて，ARM プロセッサにおける実行サイクル数を概算し，DRMT によるおおよその性能向上を示した．こちらも Dalvik レジスタにマップ可能な物理レジスタ数を 2 ~ 6 まで変化させ，DRMT の構成はフルセットアソシアティブ構成とした．

9.1.2 MIPS プロセッサにおける DRMT の効果

9.1.2.1 Dalvik レジスタ数が少ないメソッドの場合

デコーダが生成した MIPS 命令の割合を図 9.1 に示す．グラフの横軸はプログラム名および DRMT の構成，縦軸は生成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（DRMT 無効），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 によって生成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，総生成命令数に対し，生成されたロード/スト

表 9.1: 評価に用いたプログラムとパラメータ

プログラム	内容	パラメータ
Sieve	エラストテネスのふるい	探索する範囲は 1,200,000.
Matrix	正方行列の行列積	行列の大きさは 128.
Irrm	非正方行列の行列積	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.

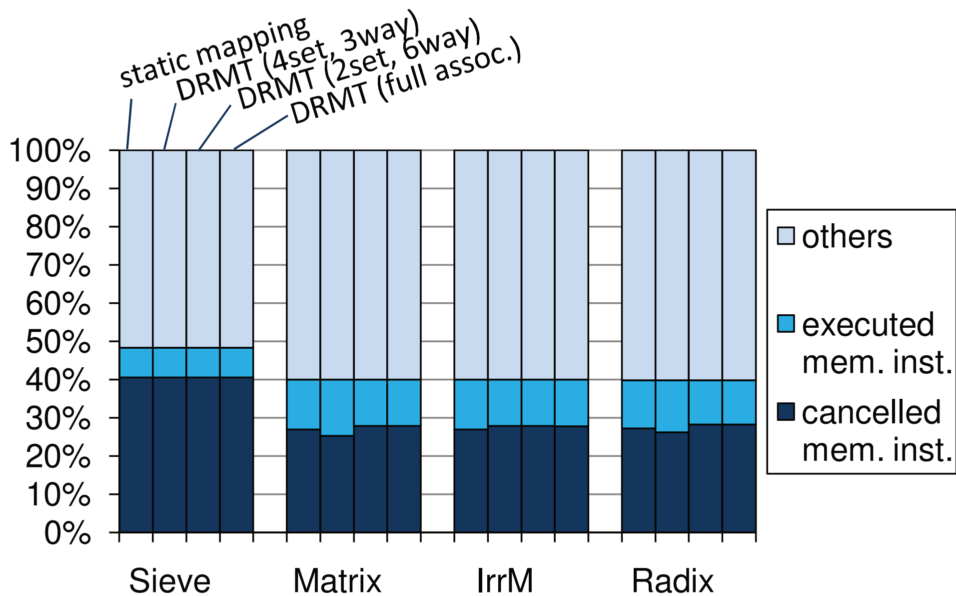

図 9.1: デコーダが生成した命令の内訳 （MIPS で int 型のプログラムを実行した場合）

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 9.2 に，DRMT を用いない場合に生成されたロード/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．構成は MIPS int 型 DRMT12 エントリ フルアソシアティブである．プログラム毎に並んだ 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わかる．

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．

9.1.2.2 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．

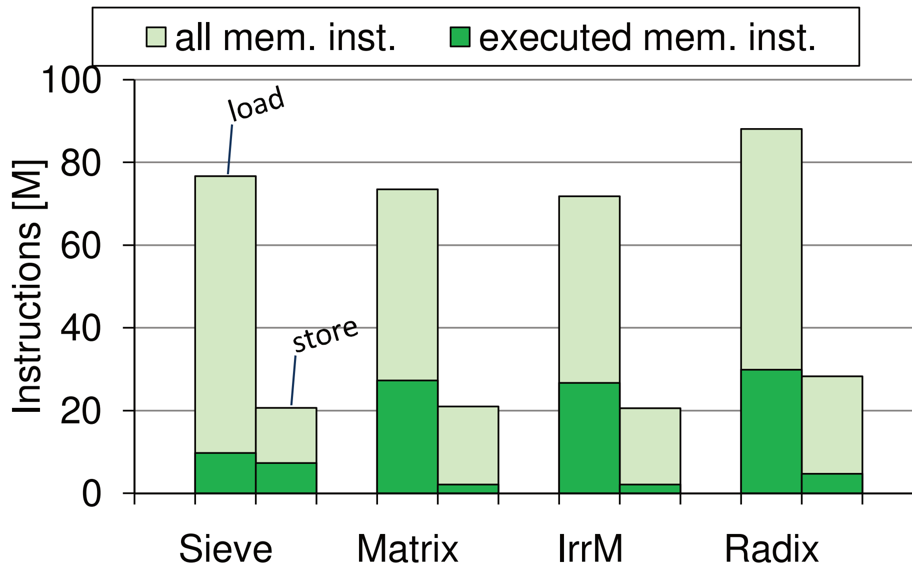

図 9.2: 生成されたメモリ・アクセス命令数（左:ロード 右:ストア）

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，デコーダが生成した MIPS 命令の割合を図 9.3 に示す．グラフの見方は図 9.1 と同様である．

総生成命令数に対し，生成されたロード/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 構成に関わらず，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 ウェイのほう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．一旦 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．

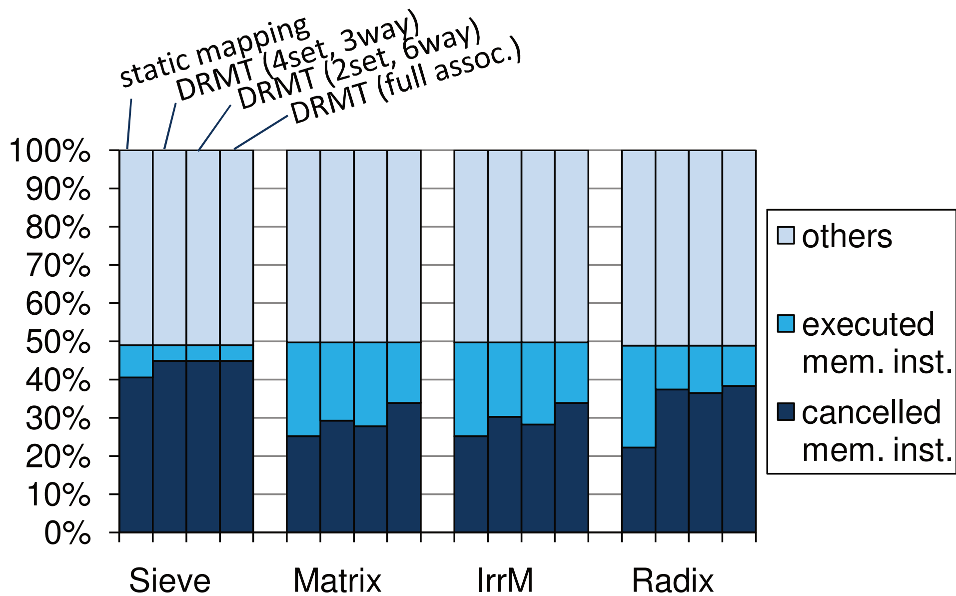

図 9.3: デコーダが生成した命令の内訳（MIPS で long 型のプログラムを実行した場合）

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．

9.1.3 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 9.1.2.1 で用いたプログラムを使用する．

デコーダが生成した ARM 命令の割合を図 9.4 に示す．グラフの横軸は Dalvik レジスタ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 3.1）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．

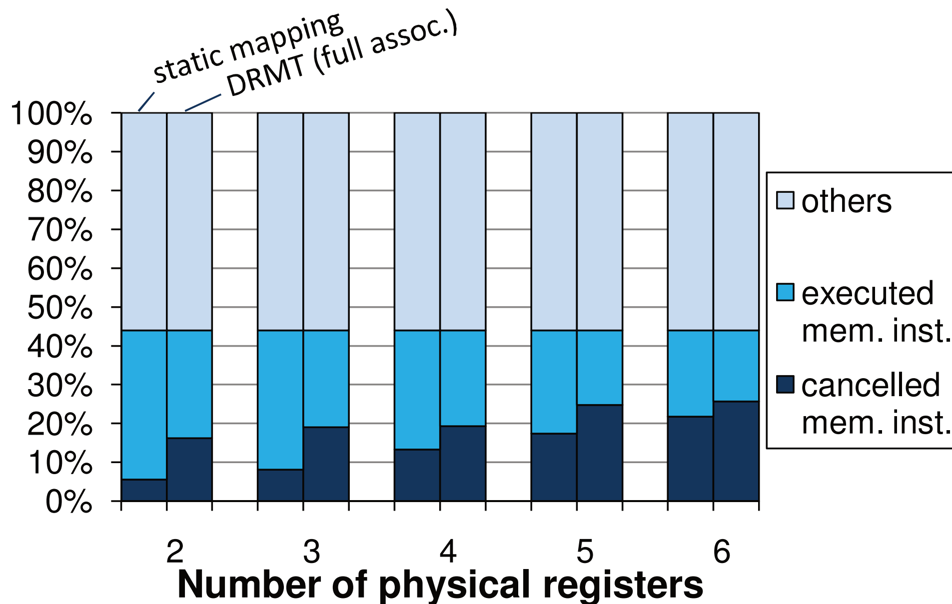

図 9.4: デコーダが生成した命令の内訳（ARM で int 型のプログラムを実行した場合）

9.1.4　実行サイクル数を基準とした概算比較

これまでの比較はデコーダが生成した命令数ベースの比較である．現段階で正確にサイクル数を求められる Dalvik アクセラレータの実装はない．したがって実行サイクル数を基準に比較するにあたり，命令種別毎に必要なサイクル数を定義し概算で比較した．以下に各命令種別毎にサイクル数の定数を示す．

実行を回避したロード・ストア命令 命令が生成されないことから，0 サイクル．

実行したロード・ストア命令 メモリへのアクセスが生じることから，100 サイクル．

その他のデコーダが生成した パイプラインで通常実行されるとして，1 サイクル．

現代のプロセッサにおけるメモリアクセスは，複数段からなるキャッシュからのアクセスの成否によりサイクル数が変動するが，今回は常に主記憶へのアクセスが発生するものとして算出した．

図 9.5 に節 9.1.3 と同一の構成，ARM における int 型 Sieve プログラムを実行した場合の，実行サイクル数比率を示す．各プログラムは 100% がデコーダの生成した命令をそのまま実行した場合である．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．縦軸の割合は実行サイクルの比率を示す．上から順に，DRMT によって実行を回避できたロード/ストア (reduced)，回避できず実行したロード/ストア (executed mem. inst)，ロード/ストア以外の命令 (others) 実行サイクル数について比率を表す．すなわち，後者 2 つの合計が，DRMT を適用し各構成で実行されたサイクル数となる．

サイクル数を基準に概算を比較すると，Dalvik レジスタのロード/ストアによるメモリアクセスのボトルネックが顕著に表れる，その他の命令の実行サイクル (others) が占める率はいずれも 1% 未満となった．

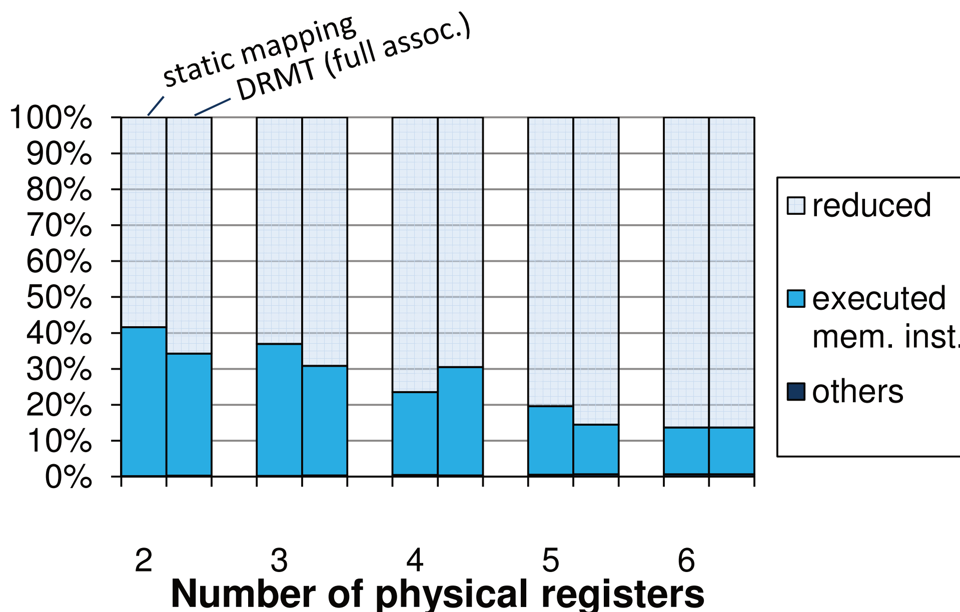

図 9.5: 実行サイクル数の削減 （ARM で int 型の Sieve プログラムを実行した場合）

DRMT 適用時は，使用できる物理レジスタが 2 個に限定されていても，サイクル数を 65.7% 削減した．物理レジスタ 4 個の場合において，DRMT のサイクル数削減効果が静的マッピングに比べて低下しているが，今回の評価においては，使用可能な物理レジスタが増えるにつれて削減したサイクル数は物理レジスタ 6 個構成において 86.3% まで上昇した．

使用可能なレジスタの本数が増えるにつれて，静的マッピングとの削減サイクル数の比率は狭まってゆく．よって DRMT は Dalvik レジスタの保持先に割り当てられるレジスタの本数が限定されたアーキテクチャにおいてより高い効果が得られる．

9.2 デコーダが生成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 デコーダが生成する命令列は，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，デコーダが生成する MIPS 命令列を，Android2.2 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，元のバイトコードの 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，それとデコーダが生成する命令列とを比較する．

9.2.1 デコーダが生成するコード

9.1.2.1 で使用した Sieve プログラムのバイトコード列を表 9.2 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 命令を生成する（表の「DRMT 無し」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，生成される命令数は 15 命令（60 バイト分）にまで縮小する．

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 命令のみ生成される．MIPS の add 命令 1 命令は，元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．例えば，boolean 型の配列へ書き込む aput-boolean 命令は 10 個（40 バイト分）の MIPS 命令を生成する．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．

表 9.2: 評価コード Sieve からデコーダが生成する MIPS 命令のサイズ

		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	48	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 等）	-	-	16	0
	合計	92	60	136	88

[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．

9.2.2 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，デコーダが生成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 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，(トレースした JIT 対象のバイトコードが分岐で終わるなど）ネイティブ・コードへ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.

の手順を踏まえ，例外処理を開始する．一方，デコーダの例外判定は，

1. 例外の発生を判定.
2. 発生した例外に対応する，例外要因の定数をレジスタ $ESTAT にセット．JRCM 命令を生成，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．

デコーダの生成するコードに比べ，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 が生成するコードよりも膨張してしまう．表 9.2 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．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．

第10章　結論

本論文では，Dalvik VM の性能が低いことに着目し，その性能を改善する手法として，Java バイトコードにおける Jazelle DBX に代表されるプロセッサ・パイプラインに搭載する形式の，Dalvik バイトコード・アクセラレータを提案し Dalvik VM においても同様に Dalvik バイトコードの実行を高速化が可能であることを示した．この結果を踏まえ，得られた成果と知見を述べる．

最後に今後の展望として，2 点挙げる．一つは本論文で示すことのできなかった，今後取り組むべき実装と評価について示す．もう一つは本論文で実装したロード・ストアの削減手法について，異なる改善手法を提案する．

10.1　研究成果

10.1.1　Dalvik アクセラレータの実装方法

本研究における Dalvik バイトコード・アクセラレータは，従来 Java においてハードウェア高速化手法として Jazelle 方式を基に手法を提案した．Jazelle 方式は実装の基本方針である，プロセッサ・パイプラインにバイトコードからプロセッサ・ネイティブな命令を生成するデコードステージを追加する点，実行モードを遷移する命令の存在は明らかであったが，それ以上の詳細は ARM 社より公にはなっていない．

そこで本研究では，バイトコードからプロセッサ・ネイティブな命令を生成する，デコーダの内部設計とその動作を示し，Dalvik バイトコードに対して適用した．そして Dalvik アクセラレータが Dalvik バイトコードから生成するプロセッサ・ネイティブな命令列とその手順を示した．

10.1.2　Dalvik アクセラレータによる命令生成

本研究で実装した Dalvik アクセラレータは，インタプリタに比べて，より MIPS 命令の実行数が少ない命令生成ができることが明らかとなった．従来インタプリタが行っていた命令のフェッチについては，フェッチステージで行われるため，削減されるのは自明であった．更に，プロセッサ側に若干命令を追加することで，よりインタプリタに比べて効率の高いコード生成が可能となった．

その例として，例外の検知が挙げられる．インタプリタでは，これらの検知処理を複数の MIPS 命令を実行させせることで検知，必要であれば例外の発行を行っていた．一方 Dalvik アクセラレータでは，プロセッサへの命令追加により生成される命令は少なくなり，プロセッサ側で判定，例外の発生と実行モードの復帰まで可能となる．

10.1.3 DRMT によるデコード命令の削減

バイトコードをありのままに解釈し実行すると，ハードウェア・アクセラレータ，ソフトウェア・インタプリタ限らず，演算対象となるレジスタをロードし，演算結果をバイトコード毎にストアしていた．バイトコードの都度ロード・ストアを行うのは効率が悪い．本論文では DRMT を提案した．実行されるバイトコード間で複数回利用される，Dalvik レジスタを物理レジスタにマッピングする．そして物理レジスタにマップした Dalvik レジスタがあるのならば，ロード・ストアを削減する機構を提案した．

本論文の評価ではバイトコードよりデコードされる命令から，DRMT を用いることで 60%以上，最大 87%のロードを削減，65%以上，最大 90%のストアを削減できることを示した．

サイクル数ベースでの性能向上の効果は，正確にサイクル数を算出可能な実装が現時点で存在しないことから，命令数ベースの削減結果と，事前に定義した命令種別毎のサイクル数より，概算した．サイクル数は命令数ベースに比べ，メモリアクセスの回数が大きく計算結果に影響を及ぼす．結果，ARM アーキテクチャの構成で，用いることのできる Dalvik レジスタが 2 個と限定された環境下においても， DRMT により実行サイクル数を 3 分の 1 に削減可能な例を示した．

続いて，Android2.2 より Dalvik VM に実装された JIT との間で，生成される命令の比較を行った．この比較においては，JIT 側についても生成されたネイティブ・コードから，更に手計算で Dalvik レジスタのロード・ストアを最大限に削減，最適化された状態で比較を行った．結果，DRMT は JIT よりも少ない MIPS 命令列を生成し，効率的な命令生成が行われることを確認した．

12 バイトの Dalvik バイトコードより，DRMT が全ヒットした Dalvik アクセラレータは 60 バイトの MIPS 命令列を生成するのに対し，最大限の最適化を行った JIT コードでは 88 バイトとなった．この結果より JIT コードに比べても，DRMT が有効な Dalvik アクセラレータは効率的な MIPS 命令列のデコードが行われていることを明らかにした．

10.2 今後の課題

10.2.1 SimMips への Dalvik アクセラレータ実装と評価

今回，SimMips 上において Android を実行でき，その上で動作する性能が妥当になるよう変更を加えた．しかし Dalvik アクセラレータを SimMips に実装することは完了していない．よって今回の評価は限定的なものとなった．DRMT の削減効果の評価は Dalvik デコーダのシミュレータを利用して評価した．JIT とのコード品質の比較では，実際に生成した命令でなく，命令ジェネレータと DRMT のシミュレータの実行結果を用いた．

実行モードの遷移が存在するバイトコードも含めた評価には，SimMips への Dalvik アクセラレータ搭載，そして Dalvik VM の Dalvik アクセラレータ対応の実装が必要である．実行モードの遷移や VM 内部の挙動について，現在用いている命令ジェネレータと DRMT のシミュレータを用いて再現，評価を行うのは困難である．

SimMips へ Dalvik アクセラレータを統合し，Dalvik VM 上で実行モードの遷移が可能であれば，評価方法が限定されなくなる．また，実行モードの遷移に伴うオーバヘッドの評価も可能となる．

10.2.2　ハードウェアでの実装

Dalvik アクセラレータの実装方法として，シミュレータだけでなく，ハードウェアによる実装も考えられる．今回，確認できる範囲では，Android を起動可能で，Dalvik アクセラレータを実装するために改造可能な， HDL によるプロセッサ実装を用意することはできなかった．Dalvik アクセラレータを搭載可能な HDL 実装を調査，もしくは開発し，ハードウェアによる Dalvik アクセラレータを実現するのが望ましい．

ハードウェアによる Dalvik アクセラレータの実装があれば以下の評価手法が行える．より Dalvik アクセラレータの有効性を細かく評価できることとなる．

10.2.2.1　ハードウェア量の変化

Dalvik アクセラレータは既存のプロセッサの内部にアクセラレータを組み込むことから，回路の規模を示す論理ゲート数は少なからず増加する．今回実装の基とした Jazelle DBX は 3.3 にて示したように，少ないゲート数でハードウェア・アクセラレーションが行えることを特徴としている．アクセラレータを組み込み機器向けに搭載するにあたっては，回路規模と規模当たりの性能向上率は重要な指標となる．

アクセラレータが使用する論理ゲート数は，アクセラレータがデコード可能なバイトコードの命令数，DRMT の構成，実装の有無によって変化すると考えられる．性能の向上と回路規模の増大の関係から，効率的なハードウェア・アクセラレータに求められる設計を明らかにできるものとみられる．

また，プロセッサに Dalvik アクセラレータへ遷移するための命令を追加しなければならないため，この必要な動作を加えた，プロセッサの回路規模の増分も評価しなくてはならない．Dalvik アクセラレータ本体同様に，極力最小に抑えるべきである．

10.2.2.2　実際のアプリケーションによる評価

現在の SimMips を利用した評価では，Android システムに大きな変更を加えており，コンソールから Dalvik VM を呼び出すことで評価を行っている．評価可能なプログラムは，Java の基本的なクラスを用いたものであり，その結果の取得手段にしても Android ログシステムや，System.out.println() メソッドを用いるなど，限定されている．

これらの制約から，評価に用いるプログラムが限定される．今回評価に用いたプログラムも，ベンチマークプログラムや短いコードが多くを占める．小規模なコードによる評価は，JIT の評価結果に大きく影響を及ぼす．トレースベース JIT でありながら，プログラムの大半が JIT 対象になる，同一のバイトコードのトレースに対し複数回の JIT が発生するなど，実際のアプリケーション上において現実的でない挙動を示した．

通常の Android の構成として起動するには，プロセッサだけでなく，ストレージやフレームバッファ，入力インタフェースといった周辺ハードウェアの実装も必要である．しかしこれらのハードウェアの実装が揃い，通常の Android の構成で起動させられれば，実際のアプリケーション上で評価を行うことができる．

10.2.3　効率的な Dalvik レジスタのマッピング手法の考案

Dalvik アクセラレータの発行するロード・ストアの命令削減手法として，今回 DRMT を提案し，その削減効果を示した．DRMT は LRU によるリプレースポリシーを持ち，比較的複雑なテーブルを有する．回路規模の評価が行えないため，プロセッサならびにアクセラレータに占める面積は未知数である．

しかし，より小規模となるマッピング手法は求められる．DRMT よりも簡素な構造のマップ方法として，「セミスタティック・マップ」が提案する．これは，命令ならびに対象とするオペランドによって，ロードされた Dalvik レジスタの再利用性が異なるに注目し，物理レジスタにマップする Dalvik レジスタを限定するものである．

例えば，演算命令の宛先となる Dalvik レジスタは，そのあと別の演算や制御に再利用される可能性が高い．このような目的の Dalvik レジスタについて物理レジスタにマップして，バイトコード間で再利用できるようにする．一方，配列操作命令における配列の参照など，オブジェクトを対象とする Dalvik レジスタは頻繁に再利用されないと判断し，一時レジスタにマップ，そのバイトコード内でのみ有効とする．

Dalvik レジスタは，メソッド中のローカル変数であり生存期間が長くない．また，図 4.2 で示したようにメソッド毎の Dalvik レジスタの使用本数が少ないメソッドが多い．これらのことから，物理レジスタの Dalvik レジスタのマップ領域が埋まることは多くないとのみられ，リプレース機構は実装しない．先着順のマップ方式とする．

評価の方法は命令の種別やオペランド毎に，Dalvik レジスタへのマップポリシーを切り替える．そして Dalvik レジスタを物理レジスタへマップした命令から後続する命令で，物理レジスタにマップした Dalvik レジスタが再利用され，ロード・ストアが削減されているかどうかで評価する．

謝辞

はじめに，研究室において，また単位認定退学後も引き続き，本研究の機会を与えて下さった，東京農工大学 大学院 工学研究院 先端情報科学部門 中條 拓伯 准教授に深く感謝致します.

本研究につきまして，技術面での議論から論文の執筆に至るまで，多岐に渡りご指導くださりました，東京大学 大学院 情報理工学系研究科 三輪 忍 助教に深く感謝致します.

本研究テーマに際しまして，Dalvik VM のインタプリタ・JIT コード生成部の分析，Dalvik バイトコード・アクセラレータのハードウェア実装と，本研究の実現性を評価する手段を多岐に渡り形にし協力を頂きました，狭山ヶ丘高等学校 吉實 大輔氏に深く感謝致します.

本研究テーマに理解を示し，より高度なレベルでのバイトコードのハードウェア・アクセラレーションの実現に向け研究を継続しています，東京農工大学 大学院工学府 情報工学専攻 小池 恵介氏，工学部 情報工学科 老子 裕輝氏に深く感謝いたします.

編入並びに研究を取り組む進路選択のきっかけであり，また長期に渡り中條研究室においてご指導くださりました，任天堂株式会社 小笠原 嘉泰氏に深く感謝いたします.

中條研究室の事務を担当し研究活動における事務手続きの協力を数多く頂きました，東京農工大学 工学部 情報工学科 中條研究室 元秘書 大澤 彰子氏，現秘書 沖田 明子氏に深く感謝します.

研究テーマやその方向性の，議論の機会やアドバイスと，研究の尽力にご支援を頂きました中條研究室の皆様に深く感謝致します.

最後に，在学中の研究活動，そして仕事と研究の両立を支えてくださりました，両親に深く感謝をいたします.

2013 年 9 月

参考文献

[1] 社団法人電気通信事業者協会. 事業者別契約数 (2013 年 3 月末現在), Mar 2013.

[2] Gartner, Inc. Gartner Says Worldwide Mobile Phone Sales Declined 1.7 Percent in 2012, Feb 2013.

[3] Google, Inc. Android. http://www.android.com/.

[4] Google, Inc. Android Developers. http://developer.android.com/.

[5] 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, 2010-04.

[6] 携帯，海外へ再進出 スマートフォンで開拓 NEC，豪などで 400 万台計画富士通，欧中印に供給. 日本経済新聞, Feb 2011.

[7] Gartner, Inc. Gartner says worldwide mobile phone sales grew 35 percent in third quarter 2010; smartphone sales increased 96 percent, Nov 2010.

[8] Google, Inc. Google TV. http://www.google.com/tv/.

[9] Android 3.0 Platform Highlights. http://developer.android.com/about/versions/android-3.0-highlights.html, Jan 2011.

[10] Android 4.0 Platform Highlights. http://developer.android.com/about/versions/android-4.0-highlights.html, Oct 2011.

[11] Open Embedded Software Foundation. http://www.oesf.jp/.

[12] Open Embedded Software Foundation. Embedded Master Developers. http://developer.oesf.biz/em/developer/.

[13] Open Embedded Software Foundation. What is OPB? http://developer.oesf.biz/em/developer/sdk/WhatIsOPB.html.

[14] 道本健二. 組み込み向け Android は 2010 年 2 月に公開 OESF がロードマップを発表. 日経エレクトロニクス, Vol. 1013, p. 33, 09 2009.

[15] Google, Inc. Android Open Source Project. http://source.android.com/.

[16] Patrick Brady. Anatomy & Physiology of an Android. In *Google I/O 2008*, 2008.

[17] 川崎進一郎. SuperH アーキテクチャ向け Android の開発. 日本 Android の会 2009 年 7 月のイベント, 2009.

[18] ppcdroid. http://code.google.com/p/ppcdroid/.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[20] D. Bornstein. Dalvik VM internals. In *Google I/O Developer Conference*, 2008.

[21] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*. USENIX Association, 2001.

[22] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 180–195. ACM, 2001.

[23] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: java for applications a way ahead of time (wat) compiler. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*. USENIX Association, 1997.

[24] Myriad Group AG. Myriad Dalvik Turbo. http://www.myriadgroup.com/Device-Manufacturers/Android-solutions/Dalvik-Turbo.aspx.

[25] Google, Inc. Multiple APK Support — Android Developers. http://developer.android.com/google/play/publishing/multiple-apks.html.

[26] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In *In Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*, pp. 153–163, 2005.

[27] 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.

[28] 太田淳, 三輪忍, 中條拓伯. 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. 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, pp. 115–132, May 2011.

[29] A. Ohta, D. Yoshizane, and H. Nakajo. Cost reduction in migrating execution modes in a dalvik accelerator. In *Consumer Electronics (GCCE), 2012 IEEE 1st Global Conference on*, pp. 502–506, 2012.

[30] 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, May 2010.

[31] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[32] 鷲見豊（訳）, Jon Mayer, Tory Downing. JAVA バーチャルマシン. O'REILLY, 1997.

[33] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*, chapter 3. Prentice Hall, 1999.

[34] Chris Porthouse. *Jazelle for Execution Environments*, 2005.

[35] Tom R. Halfill. Arm strengthens java compilers. *Microprocessor Report*, Vol. 19, No. 7, Jul 2005.

[36] Levy Markus. Java to go: Part 4. *Microprocessor Report*, Vol. 15, No. 6, Jun 2001.

[37] H. Mizuno, N. Irie, K. Uchiyama, Y. Yanagisawa, S. Yoshioka, I. Kawasaki, and T. Hattori. Sh-mobile3: Application processor for 3g cellular phones on a low-power soc design platform. In *Hot Chips 16*, 2004.

[38] Chris Porthouse. *Jazelle DBX Technology: ARM Acceleration Technology for the Java Platform*, 2005.

[39] Steve Steele. *Accelerating to Meet the Challenge of Embedded Java*, 2001.

[40] ARM Ltd. *ARM Architecture Reference Manual*, 2005.

[41] Levy Markus. Java to go: Part 1. *Microprocessor Report*, Vol. 15, No. 2, Feb 2001.

[42] Levy Markus. Java to go: the finale. *Microprocessor Report*, Vol. 15, No. 6, Jun 2001.

[43] Paul Capewell and Ian Watson. A risc hardware platform for low power java. In *International Conference on VLSI Design*, pp. 138–143, 2005.

[44] J. M. O'Connor and M. Tremblay. picojava-i: the java virtual machine in hardware. *Micro, IEEE*, Vol. 17, No. 2, pp. 45–53, 1997.

[45] H. McGhan and M. O'Connor. Picojava: a direct execution engine for java bytecode. *Computer*, Vol. 31, No. 10, pp. 22–30, 1998.

[46] Sun Microsystems. picoJava-II Datasheet, Apr 1999.

[47] Sun Microsystems. picojava-ii programmer's reference manual, 1999.

[48] Sun Microsystems. picojava-ii microarchitecture guide, 1999.

[49] W. Puffitsch and M. Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pp. 213–221. ACM, 2007.

[50] J. Kamdar. Embedded java in information appliances. Technical report, 2000.

[51] 入江直彦. 携帯電話向け Java の実行速度を 10 倍に加速する SH-Mobile 向けアクセラレータ. *RENESAS EDGE*, Vol. 7, p. 15, Oct 2004.

[52] Pendragon Software Corporation. Caffeinemark. `http://www.benchmarkhq.ru/cm30/info.html`.

[53] 内山邦男. 高性能・低消費電力マイクロプロセッサ. 日立評論, Vol. 87, No. 5, pp. 89–94, May 2005.

[54] 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.

[55] James Niccolai. Oracle sues Google over Java use in Android. Computerworld, Apr 2010.

[56] Stephen Shankland. Sun's worried that Google Android could fracture Java. CNET News, Nov 2007.

[57] Mono Project. Monodroid. `http://monodroid.net/`.

[58] Android 2.3 Platform Highlights. `http://developer.android.com/about/versions/android-2.3-highlights.html`, Dec 2010.

[59] Chris Pruett. Android でリアルタイムゲームを開発する方法: リベンジ. In *Google Developer Day 2010*, 2010.

[60] MIPS Technologies, Inc. Mipsandroid. `http://mipsandroid.org/`.

[61] 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 2008–EMB–010, pp. 23–28, 2008.

[62] MIPS Technologies, Inc. *MIPS32 Architecture For Programmers Volume I: Introduction to the MIPS32 Architecture*. 2001.

[63] 吉瀬謙二, 佐藤真平, 森谷章, 藤枝直輝, 若杉祐太, 渡邉伸平, 植原昂, 森洋介, 高前田伸也, 高橋朝英, 棟岡朋也, 山田裕介, 権藤克彦, 小林良太郎, 三好健文, 中條拓伯. MieruPC プロジェクト: 中身が見える計算機システムを構築する研究・教育プロジェクト. 情報処理学会 システムソフトウェアとオペレーティング・システム研究会コンピュータシステム・シンポジウム (ComSys2008), November 2008.

付 録A　研究業績

以下に 2013 年 5 月までの研究業績を示す.

A.1　論文誌

1. 太田淳, 三輪忍, 中條拓伯, “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”, 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, pp.115-132 (2011.5)
2. 小池恵介, 太田淳, 大島浩太, 藤波香織, 郡信幸, 竹本正志, 中條拓伯: “Android における Java アプリケーションの FPGA アクセラレーション”, 情報処理学会論文誌「組込みシステム」特集号 Vol.53, No.12, pp.2740-2751 (2012.12)

A.2　査読付き国際会議

1. Atsushi Ohta, Yoshihiro Hamada, Aira Kitamura, Noboru Tanabe, Hideharu Amano, Hironori Nakajo: “Implementation and Evaluation of Multicast Mechanism on Network Interface Plugged into a Memory Slot” The 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’07), Vol.II, pp.787-793 (2007.6).
2. Hironori Nakajo, Keisuke Koike, Atsushi Ohta, Kohta Ohshima, Kaori Fujinami: “Reconfigurable Android with an FPGA Accelerator for the Future Embedded Devices”, Proc. of the 3rd Workshop on Ultra Performance and Dependable Acceleration Systems (UPDAS2011), pp.173-178 (2011.12).
3. Atsushi Ohta, Daisuke Yoshizane, Hironori Nakajo: “Cost Reduction in Migrating Execution Modes in a Dalvik Accelerator”, Proc. 1st IEEE Global Conf.Consumer Electronics (GCCE 2012), pp.502-506 (2012.10)

A.3　査読付き国内会議

1. 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) ポスター・セッション, Vol.2010, No.5, pp.113-114 (2010.5)

2. 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 (2010.10) (情報処理学会 2010 年度コンピュータサイエンス領域奨励賞 (組込みシステム研究会) 受賞)

3. 小池恵介，太田淳，大島浩太，藤波香織，郡信幸，竹本正志，中條拓伯: “FPGA アクセラレータによる Android アプリケーションの高速化手法”, 組込みシステムシンポジウム (ESS2011) pp.10-1-10-8 (2011.10)

4. 吉實大輔, 太田淳, 中條拓伯, 三輪忍: “Dalvik アクセラレータのハードウェア実装”, 組込みシステムシンポジウム (ESS2012), pp.225-226 (2012.10)

A.4 研究会，査読なしシンポジウム，技術報告書

1. Atsushi Ohta, Yoshihiro Hamada, Akira Kitamura, Noboru Tanabe, Hideharu Amano, Hironori Nakajo: “メモリスロット直結型ネットワークインタフェースへのマルチキャストの実装と評価” 情報処理学会「数理モデル化と問題解決」研究会報告, Vol.2007-MPS-65, pp.41-44 (2007.6).

2. 太田淳，金美善，田邊昇，中條拓伯 : “DMA で主記憶をアクセスする CPU における不連続アクセスの連続化”，第 15 回「ハイパフォーマンスコンピューティングとアーキテクチャの評価」に関する北海道ワークショップ (HOKKE-2008), Vol.2008-ARC-177/HPC-114, pp.7-12 (2008.3).

3. 田邊昇，北村聡，宮部保雄，宮代具隆，天野英晴，太田淳，中條拓伯: “ハードウェアを用いたメッセージ交換システムのスケーラビリティ改善”，第 15 回「ハイパフォーマンスコンピューティングとアーキテクチャの評価」に関する北海道ワークショップ (HOKKE-2008), Vol.2008-ARC-177/HPC-114, pp.181-186 (2008.3).

4. 田邊昇, 太田淳, 金美善, 中條拓伯: “DMA で主記憶をアクセスする CPU における不連続アクセスの連続化”, 第 7 回情報科学技術フォーラム (FIT’08), RC-005, (2008).