Vol.2014-SE-184 No.3
Vol.2014-EMB-33 No.3
2014/5/19

FPGAを用いたDalvikアクセラレータの実装と評価

老子 裕輝[1]　吉實 大輔[2]　太田 淳[3]　三輪 忍[4]　中條 拓伯[1]

概要：Android 端末において，Java で記述されたアプリケーションは Dalvik バイトコードと呼ばれる中間コードに変換され，さらに Dalvik 仮想マシンを介して実行される．この VM によるコード実行は性能低下やメモリ使用量の増加といった様々な問題を引き起こすが，この問題に対して我々はハードウェアで直接中間コードを実行する Dalvik アクセラレータを提案してきた．これまで我々はアクセラレータの構造の提案や生成コードに関する予備評価を行った．本研究では新たに，Dalvik アクセラレータを実際に FPGA 上において MIPS プロセッサとともに実装し，評価を行った結果，プログラムの実行時間を大幅に削減できることがわかった．本稿では，その実装の詳細と評価結果について報告する．

キーワード：組込み，FPGA，Android，Dalvik，アクセラレータ

Implementation and Evaluation of Dalvik Accelerator Using FPGA

YUKI OIGO[1]　DAISUKE YOSHIZANE[2]　ATSUSHI OHTA[3]　SHINOBU MIWA[4]
HIRONORI NAKAJO[1]

Abstract: On an Android device, a Java application is compiled to the intermediate code called as Dalvik bytecode and then run with Dalvik Virtual Machine. Such a execution model causes performance degradation and memory wastage. In order to solve this issue, we have proposed a Dalvik accelerator which directly executes an intermediate code with specialized hardware. Previous studies by our group have shown architecture and preliminary evaluation about generated code. Therefore, in this study, we have implemented the accelerator as well as a pipelined MIPS processor on FPGA to evaluate our proposed mechanism. The result shows that the proposed mechanism significantly improves performance for some programs. Here we report detailed implementation and evaluation results.

Keywords: Embedded, FPGA, Android, Dalvik, Accelerator

1. はじめに

近年，Google 社のスマートフォン向けプラットフォームである Android を搭載した機器が急速に普及している．この従来の携帯端末にはない，快適な Web ブラウジング機能やタッチパネルによる直感的な操作などを可能にした Android は，携帯端末用 OS として海外ではトップ，日本でも 2 位のシェアを保持している [1]．また，Android は Linux カーネルをベースにしており，ソースコードが公開されているため，多くの端末メーカが独自のカスタマイズを行った上でスマートフォンに組み込んでいる．

しかし，Android アプリケーションを快適な動作速度で実行するためには，高性能な CPU や大容量のメモリが必要となる．Android アプリケーションは基本的に Java で記述され，VM を介して実行される．一般に，VM を介したアプリケーション実行はネイティブ・コードの実行に比べて低速であり，ネイティブ・コードと同等の性能を達成するためにはより高性能な CPU を必要とする．VM によ

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[2] 学校法人 狭山ヶ丘学園
Sayamagaoka Senior High School
[3] 株式会社 日立情報通信エンジニアリング
Hitachi Information & Telecommunication Engineering, Ltd.
[4] 東京大学
Tokyo University

る性能低下を抑えるため，Android では高速化手法として JIT コンパイルを採用しているが，JIT コンパイルによる実行は一般に多くのメモリを必要とするため，電力制約の厳しい携帯情報端末に向いているとはいい難い．Java アプリケーションがネイティブ・コードで記述されたアプリケーションとほぼ同等の電力効率で実行できることが望ましい．

この問題に対し，我々は Dalvik アクセラレータを提案してきた [2][3][4][5]．Dalvik アクセラレータでは，本来 Dalvik VM が行う，Dalvik バイトコードから機械語への変換を，可能な限りハードウェアで行うことにより高速化を図る．また，アクセラレータは従来のプロセッサパイプラインを拡張する形で実現できるため，演算ユニットやレジスタなど，既存のプロセッサ資源を活用可能というメリットを有する．

これまでの研究では，Dalvik アクセラレータの構造・動作に関する提案や，ハードウェア実装のための生成コードに関する予備評価を行ってきた．

本稿では新たに，Dalvik アクセラレータを組み込み，実際に Dalvik バイトコードを変換・機械語実行可能なプロセッサである DalvikCore のハードウェア実装を行った結果について報告する．また，本アクセラレータの重要な特徴である，動的レジスタマッピングテーブルとこれに基づいた冗長なロード命令削減機構を実装し，Dalvik バイトコードを高速に実行できるプロセッサとしての評価を示す．

本論の構成は以下の通りである．まず次章では，Java VM において利用されてきた従来の高速化手法および Dalvik VM の仕様や Android における現状の高速化手法について述べる．続く 3 章では，提案手法である Dalvik アクセラレータのアーキテクチャおよびこれを組み込んだプロセッサである DalvikCore について説明する．4 章では，アクセラレータ内のレジスタマッピングテーブル，DRMT（Dalvik Register Mapping Table）およびこれを用いた高速化機構について述べる．そして，5 章でバイトコード実行の高速化に関する評価を行い，6 章でまとめる．

2. 関連研究

2.1 Java VM における高速化

従来の Java 実行環境であった Java VM においては，過去 20 年近い歴史の中で多くの高速化手法が提案されてきた．これらは，主にソフトウェアによる手法とハードウェアによる手法の 2 つに分類される．

2.1.1 ソフトウェアによる高速化手法

ソフトウェアによる高速化手法として，実行時コンパイル（Just In Time，以下 JIT とする）や事前コンパイル（Ahead of Time，以下 AOT とする）が挙げられる [6][7]．これらは，プログラム中の頻繁に実行されるメソッドやループ部分を検出し，該当のバイトコードを実行時または実行前に機械語へとコンパイル，メモリ上に格納する手法である．この手法により，一度コンパイルされたバイトコードは以降，高速に実行できるが，コードサイズが膨らみ，メモリを圧迫するという欠点が存在する．これまでの研究において，JIT コンパイルによって生成される機械語のメモリ量は，バイトコードのメモリ量と比較して 11.3 倍になることが判明している [3]．

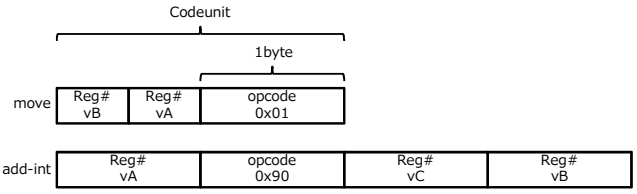

図 1　Dalvik バイトコードフォーマット
Fig. 1　Format of Dalvik bytecode.

2.1.2 ハードウェアによる高速化手法

一方，ハードウェアによる高速化手法では，その多くが Java バイトコードを直接解釈・実行可能な Java プロセッサおよびホストプロセッサに付随するアクセラレータ回路として実現されてきた．こういったアプローチは，直接実行可能な Java バイトコードの規模や実行形態が多岐に渡り，ほぼ全てのバイトコードがプロセッサで実行可能な picoJava[8][9] や，複雑なバイトコードは VM に処理を任せるアクセラレータ回路の Jazelle DBX[10][11][12] などが挙げられる．

2.2 Dalvik VM の高速化へのアプローチ

現在，Android で利用されている高速化手法として，Android2.2 以降に搭載された JIT コンパイルが特に知られている．Google は JIT コンパイルにより，Android2.1 と比較して 2〜5 倍の高速化が得られたと発表している [13]．

しかし，スマートフォンといった主記憶容量に制限のある組込み機器においては，ソフトウェアによる高速化には大きな制約がある．また，プロセッサ/アクセラレータで直接バイトコードを処理する場合，ハードウェアで実行可能なバイトコード数は概ね回路規模に比例するため，性能と設計コストのトレードオフも検討しなければならない．

このような理由から，Android における高速化は，ハードウェア/ソフトウェアの両手法に対して依然求められている．しかし，Dalvik VM には，その内部仕様において従来の Java VM と大きく異なる点がいくつか存在する．そのため，前節で述べた高速化手法をそのまま適用することは難しい．本節では，それらの相違点について示す．

2.2.1 Dalvik バイトコード

Dalvik VM が解釈・変換する Dalvik バイトコードは，独自命令セットを持つ中間コードである．**図 1** に示すコードユニットと呼ばれる 2 バイトの最小単位 1〜5 個で構成され，最長で 10 バイトの可変長命令となる．バイトコード

のオペランドは，次に述べる Dalvik レジスタを指すが，同一の演算を行うバイトコードでも，アクセス可能な Dalvik レジスタ番号の範囲が異なる命令が存在する．したがって，メモリ使用の観点から，可能な限り命令長の短いバイトコードを使用することが望ましい．

2.2.2 Dalvik レジスタ

Dalvik バイトコードの演算対象である Dalvik レジスタは，ハードウェア上のレジスタではなく，主記憶上に確保された単なる配列データである．1 つのレジスタは 4 バイト幅で，メソッドごとに 65,536 個のレジスタが使用可能と規定されている．しかし，メモリ上の仮想レジスタが演算対象であるため，プロセッサ上の物理レジスタもしくは専用のハードウェアレジスタへの値のロード/ストアが必要となる．その結果，1 つのバイトコードからはロード命令・演算命令・ストア命令がこの順番で生成され，プロセッサの実行性能に大きな影響を与えるメモリアクセスが増えてしまう．

3. Dalvik アクセラレータと DalvikCore

我々が提案する Dalvik アクセラレータは，**図 2** に示すように，プロッサパイプラインのフェッチおよびデコードステージ間に，Dalvik バイトコードを複数の機械語列へと変換するデコーダを設ける．本方式は，ARM 社の Jazelle DBX 同様の方式であり，他の Java プロセッサ/アクセラレータと比較して，性能増加量に対する回路規模の増加が比較的少なく良好であることが分かっている [14].

3.1 Dalvik アクセラレータの構造

Dalvik アクセラレータの内部構造を**図 3** に示す．アクセラレータは 5 つのモジュールおよび 3 つのテーブルから成っており，プロセッサのフェッチャから入力された Dalvik バイトコードのコードユニットを，32 ビットの MIPS 命令に変換し出力する．

本稿では，次節で述べる MIPS ソフトコアプロセッサにアクセラレータを搭載するため，アクセラレータのターゲットを MIPS 命令セットと定めている．これまでの研究により，生成される MIPS 命令数について，従来の Dalvik VM による処理では 1 バイトコードあたり平均 19.3 命令であるのに対し，アクセラレータでは平均 7.0 命令が生成され，約 6 割の命令数削減となることが判明している [15]. また，本アクセラレータは生成する命令セットを変更することにより，多様なプロセッサアーキテクチャに応用可能であることに注意されたい．

以下に Dalvik アクセラレータを構成する個々のモジュールおよびテーブルについて説明する．

3.1.1 Bytecode Buffer モジュール

本モジュールは 80 ビットのバッファであり，入力されたコードユニットの時系列から 1 つのバイトコードを形成，出力する．Dalvik バイトコードは可変長命令かつコードユニットのフェッチ幅が 16 ビットであるため，2 コードユニット以上の長さをもつバイトコードに対しては，複数回のフェッチを繰り返した後に出力する．

3.1.2 Refer Table モジュール

本モジュールでは，入力された Dalvik バイトコードからオペコードとオペランドを切り出し，以降のステージに送り出す機能をもつ．バイトコードは命令によってオペランドの位置が全く異なるため，命令の種類に応じてデコードしなければならない．

3.1.3 Conversion Table

Conversion Table は，Dalvik バイトコードから出力される MIPS 命令のひな型を保持するテーブルであり，First Instruction Table と Following Instruction Table の 2 つのテーブルから構成される．この 2 つのテーブルが保持するフィールドは大きく 2 つに分類される．1 つは，それぞれがテーブル参照のために必要となるフィールド，もう 1 つは 2 つのテーブルが共通してもつ，生成 MIPS 命令に関する Inst フィールドである．そして，2 つのテーブルはそれぞれ異なった用途で参照される．

First Instruction Table

First Instruction Table は，Dalvik バイトコードのオペコードがインデックスとして参照され，バイトコードから最初に生成される MIPS 命令のひな形を保持する．`Next#`フィールドには，2 番目に生成する MIPS 命令を Following Instruction Table で参照するためのポインタを格納する．バイトコードから複数の MIPS 命令が生成される場合，2 番目以降の命令が Following Instruction Table のどこにあるかを，このフィールドの値を基に参照する．

Following Instruction Table

Following Instruction Table では，バイトコードから生成される 2 番目以降の MIPS 命令のひな形を保持する．First Instruction Table から出力されたポインタを基に，2 番目に生成する MIPS 命令を参照する．以降は，そのエントリの `Next#`フィールドを基に，Following Instruction Table を繰り返し参照する．生成する最終命令の `Next#`フィールドは 0 となる．

Conversion Table に格納されている各 Inst フィールドの値はひな形であるため，変換中のバイトコードのオペランドを反映した値に修正する必要がある．最終的に MIPS 命令の各フィールドに格納される値は次の 4 種類であるが，これらの内，Dalvik バイトコードのオペランドに関する後者 3 つについては Get Operand モジュール以降で上書き修正される．

- Conversion Table 内の即値
- 即値としてのバイトコードオペランド
- オフセット利用としてのバイトコードオペランドが指し示す Dalvik レジスタ番号

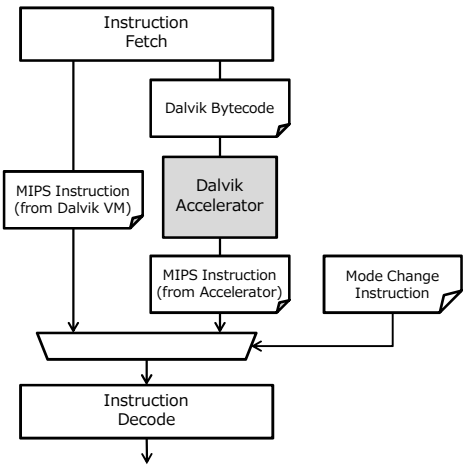

図 2 Dalvik アクセラレータによるバイトコード実行
Fig. 2 Bytecode execution by Dalvik accelerator.

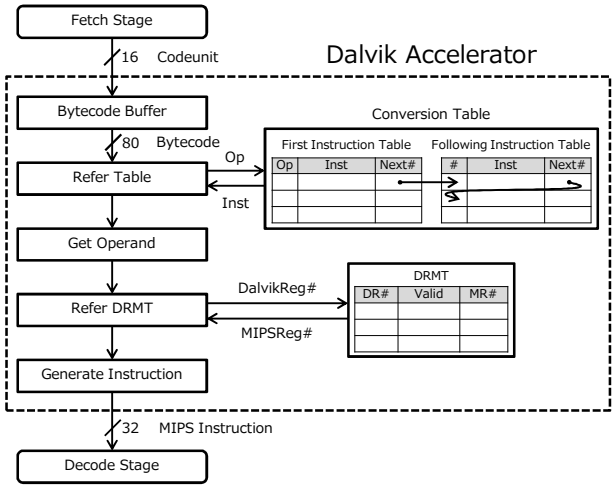

図 3 Dalvik アクセラレータのモジュール図
Fig. 3 Module diagram of Dalvik accelerator.

First Instruction Table

Bytecode Op	Inst					Next#
	format	opcode	···	shamt	immediate	
0x00						
⋮						
0x90	I	lw		-	second operand (as offset)	10
⋮						
0xFF						

Following Instruction Table

Number	Inst					Next#
	format	opcode	···	shamt	immediate	
0						
⋮						
10	I	lw		-	third operand (as offset)	11
11	R	add		-	-	12
12	I	sw		-	first operand (as offset)	0

図 4 変換テーブル
Fig. 4 Conversion Table.

- バイトコードオペランドが指し示す Dalvik レジスタがマッピングされた MIPS レジスタ番号

3.1.4 Get Operand モジュール

本モジュールでは，バイトコードオペランドの値を即値として MIPS 命令で使用する場合に，rs，rt，rd，immediate フィールドを修正する．例として，Dalvik レジスタへの即値転送バイトコードである `const/16` から生成される MIPS 命令には，addi 命令が含まれる．このとき，バイトコード中の 16 ビットの即値を addi 命令中の immediate フィールドで使用する場合に当てはまる．

3.1.5 DRMT（Dalvik Register Mapping Table）

メモリ上の仮想レジスタである Dalvik レジスタは，演算時に一度ハードウェアレジスタに値をロードしなければならず，バイトコード実行の度に多数のロード/ストア命令が生成されてしまう．しかし，次章で詳細に述べるが，2 つのレジスタ間のマッピングテーブルを設けることで，この冗長性を排除可能である．

マッピングテーブルの実装方法として，規定上限数である 65,536 個の Dalvik レジスタ全てを保持できるハードウェアレジスタを考える．$32[\text{bit}] \times 65,636 = 256[\text{KB}]$ ものサイズのレジスタファイルは，一般的な組込み機器向けの L2 キャッシュメモリサイズ並みであり，このように巨大なレジスタファイルをプロセッサ内部に設けるのは現実的ではない．そこで，少容量のキャッシュを用いてマッピングテーブルを実現することとした．

DRMT では，Dalvik レジスタの番号が入力された際，マッピングテーブルを確認し，既に物理レジスタとのマッピングが存在するならば対応する MIPS レジスタ番号を，そうでなければ新たにマッピングを行った MIPS レジスタ番号を出力する．

3.1.6 Refer DRMT モジュール

本モジュールでは，DRMT から入力された MIPS レジスタ番号を使用し，MIPS 命令の rs，rt，rd フィールド値を，バイトコードのオペランドレジスタがマッピングされた MIPS レジスタに上書き修正する．

3.1.7 Generate Instruction モジュール

Generate Instruction モジュールでは，フィールドごとに分解して渡されてきた MIPS 命令を，フォーマットに基づいて 1 つのビット列に組み上げ，出力する機能をもつ．また，このモジュールはロード・ハザードの検知を行い，ストール信号を生成する機能も持つ．

3.2 MIPS プロセッサとの接続

これまでの研究では，Dalvik アクセラレータの提案と

表 1　Dalvik モード時の MIPS レジスタ割り当て
Table 1　Register mapping in Dalvik mode.

レジスタ	名称	使用方法
$0	zero	常時 0（MIPS の仕様）
$1	at	マクロ命令が一時的に使用
$14-$15	SCRH0/1	MIPS 命令が利用可能な一時レジスタ
$16	PC	Dalvik バイトコードの PC
$17	FP	Dalvik レジスタポインタ
$18	GLUE	Dalvik VM 資源ポインタ
$19	EHND	VM のハンドラ・アドレス
$20	INST	バイトコードのベースアドレス
$21	EOBJ	VM へ戻る要因のオブジェクト参照
$22	ESTAT	VM へ戻る要因を収めたステータス値
$23	EPC	VM へ戻る要因のバイトコードの PC
$24-$25	SCRH2/3	MIPS 命令が利用可能な一時レジスタ
$26-$31	-	カーネル・MIPS プログラムが使用

単体での試作に留まり，アクセラレータを組み込んだプロセッサの実装は行われていなかった．本稿では，既存の MIPS ソフトコアプロセッサに対して Dalvik アクセラレータを組み込み，バイトコードの変換から MIPS 命令の実行までを可能とした 1 つのプロセッサ，DalvikCore として実装を行った．

アクセラレータを組み込むプロセッサは，東京工業大・吉瀬研究室が公開している MIPS ライクなソフトコアプロセッサ，MipsCore を利用する [16]．本プロセッサは，MIPS32Release2 に準拠した約 100 種類の命令セットをもち，HDL で論理合成可能なソフトコアプロセッサである．

MipsCore を利用した理由として，第 1 に Android がサポートしているアーキテクチャである ARM，MIPS，x86 の内，わかりやすい命令セットをもち改編が容易な MIPS アーキテクチャであること，第 2 に MipsCore が GPL ライセンスの下，HDL コードが公開されているパイプライン化された MIPS ソフトコアプロセッサであることを挙げる．

3.2.1　2 つの実行モード

DalvikCore は 2 つの実行モードを有する．1 つは，変換可能なバイトコードを複数の機械語に変換して高速に実行を行う Dalvik モードである．もう 1 つはアクセラレータで処理不可能なバイトコードがフェッチされた場合に，Dalvik VM へと処理を移して通常のインタプリタ実行を行う MIPS モードである．これらのモードは，後述する 2 つの命令によって相互に変更される．

3.2.2　モード間のレジスタ共有

DalvikCore では，モード変更時のレジスタ値退避に伴うオーバヘッドを軽減させるため，MIPS モードにおいて機械語インタプリタが予約済みの MIPS レジスタの一部を，Dalvik モード時に共有する．**表 1** に示したレジスタ以外の，$2 から$13 までの 12 個のレジスタが DRMT において Dalvik レジスタのマッピングに使用される．

3.2.3　MIPS 命令セットへの追加命令

DalvikCore では，アクセラレータを制御するため，既存の MIPS 命令セットに対していくつかの命令を追加している．これらの命令はモード変更命令を除き，通常の MIPS モードでは使用されることはなく，Dalvik モード時専用の命令となる．

JRCD（Jump Register and Change Dalvik bytecode mode）命令

JRCD 命令は Dalvik モードへ遷移するための命令であり，MIPS 命令における JR 命令と同様に使用される．JRCD 命令のオペランドには，アクセラレータで実行を開始する Dalvik バイトコードのアドレスを格納したレジスタを指定する．

JRCM（Jump Register and Change Mips mode）命令

JRCM 命令は MIPS モードへ遷移するための命令である．プロセッサパイプラインにおいて，プロセッサ例外や Java 例外，アクセラレータが変換できないバイトコードをフェッチしたときに，アクセラレータによって生成される．JRCM 命令のオペランドには，MIPS モードで実行を開始する MIPS 命令のアドレスを格納したレジスタを指定する．

例外チェック命令

Dalvik モードにおいて，バイトコードが例外を発生させる可能性が存在する場合，生成 MIPS 命令列の中で実際に例外が発生するかどうかをチェックする必要がある．ヌル参照チェック命令，ゼロ除算チェック命令，配列添え字番号チェック命令などを用い，例外の発生が明らかになった場合，所定のレジスタに要因や該当するバイトコードの PC をセットし，JRCM 命令を発行した上で MIPS モードに切り替える．

パイプライン制御命令

条件分岐バイトコードは生成 MIPS 命令列の最後に，条件判定命令，分岐用パイプライン制御命令、PC 変更命令の 3 つを必ずこの順に生成する．実際の PC 変更命令は MIPS の即値加算命令であるため，条件判定命令で成立/不成立を検証した後，分岐用パイプライン制御命令を用い，成立の場合は PC 変更命令より後の投機的投入されていた命令を，不成立の場合は PC 変更命令をフラッシュする．

DalvikCore は**図 5** に示す変則的なプロセッサパイプラインをもつ．図 5 中の b_n および m_n はそれぞれバイトコードとバイトコードから生成される MIPS 命令を表している．コードユニットのフェッチに必要なクロック数はバイトコードの種類に左右される他，Dalvik アクセラレータステージの通過には最低 5 クロック必要となる．それ以外のステージは MIPS モードと同様に全て 1 クロックで処理可能である．

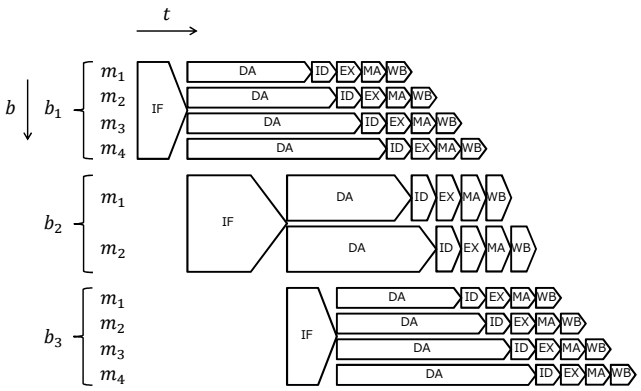

図 5　DalvikCore の動作パイプライン
Fig. 5　Pipeline of DalvikCore.

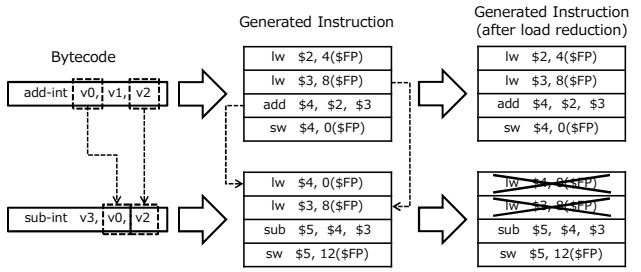

図 6　冗長なロード命令の削減
Fig. 6　Reduction of redundant load instruction.

4. DRMT による冗長なロード命令の削減

前述したとおり，単純に Dalvik アクセラレータでバイトコードから MIPS 命令を生成した際，多くのロード/ストア命令が生成されてしまう．しかし，**図 6** に示すように，同一の Dalvik レジスタをオペランドとして使用するバイトコードが連続する場合を考える．前者のディスティネーションレジスタが後者のソースレジスタとなっている場合や，両者で同一のソースレジスタを使用している場合，既に物理レジスタに値が存在しているにも関わらず再度ロード命令を行うのは非効率的である．

そこで，Dalvik レジスタと MIPS レジスタのマッピングテーブルである DRMT と，これに基づいた冗長なロード命令削減を行う機構を設ける．DRMT の実装方法として，次の 2 種類が挙げられる．

静的マッピング

これまでの研究におけるアクセラレータの試作では，DRMT は静的マッピングによる実装であった．これは，一般的な Android アプリケーションにおいて，Dalvik レジスタの使用数が同一メソッド内において高々十数個程度に収まるという前提 [14] を基に，Dalvik レジスタの 0 番から 11 番を MIPS レジスタの\$2 から\$13 までに固定して割り当てる仕様であった．しかしこの手法では，範囲外の Dalvik レジスタが使用される場合にマッピングが不可能という問題がある．

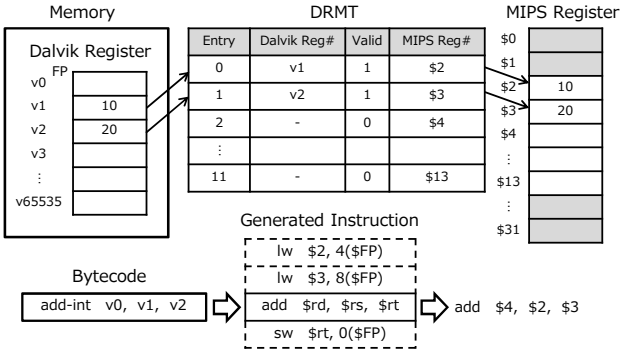

図 7　動的割り当ての DRMT
Fig. 7　Dynamic mapping DRMT.

動的マッピング

そこで，本稿では新たに動的マッピングとしての DRMT を実装した．**図 7** に示すように，動的 DRMT は 12 個のエントリを持つテーブルである．MIPS 命令の生成中，マッピングされた Dalvik レジスタ番号が入力されたならば，該当する MIPS レジスタ番号を，そうでなければ逐次的に MIPS レジスタへの割り当てを行う．ただし，演算を正常に行うために，同一バイトコードからの生成命令間では，Dalvik レジスタと MIPS レジスタのマッピングの変更を回避しなければならない．

実際にロード命令の削減は，図 3 の Refer Table モジュールにて行われる．Refer Table モジュールと DRMT はワイヤ接続されており，Conversion Table を参照すると同時に DRMT を参照し，バイトコードのオペランドレジスタが既にマッピングされているかどうかを確認する．もし DRMT のエントリにヒットするならば，該当するロード命令を生成する必要はなく，次の MIPS 命令の生成に移ることが可能である．削減可能なロード命令数は 1 バイトコードにつき最大 2 命令であるが，多くの Dalvik バイトコードは 2 個または 1 個の Dalvik レジスタをソースレジスタとするため，本機構による実行性能への影響は大きいと考えられる．

5. 評価

RTL シミュレーションを行い，Dalvik モードにおいてバイトコードで記述されたプログラムを直接実行する．このとき，動的マッピングの DRMT に基づいたロード命令削減効果の有無によるバイトコード実行の高速化率を評価する．また，DalvikCore を FPGA 上に実装する際の回路規模を評価する．

5.1 評価環境

評価環境を**図 8** に示す．バイトコードを格納した命令メモリを BlockRAM，データメモリを DDR2SDRAM とする．データメモリにアクセスするためのメモリコントローラを別途作成し，配置配線を行った後に算出される最

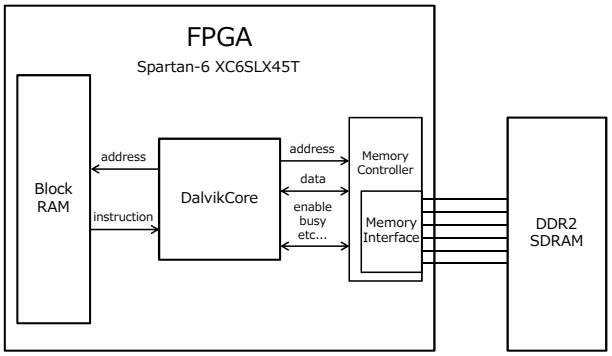

図 8　評価環境
Fig. 8　Evaluation system.

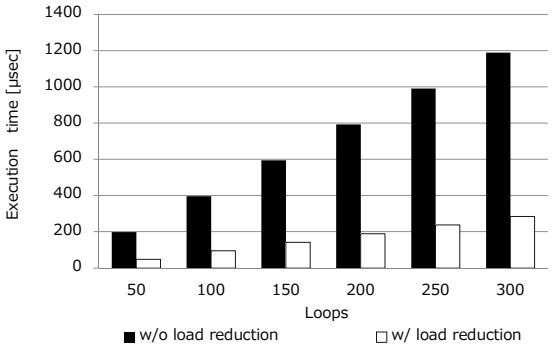

図 9　プログラム 1 の実行時間
Fig. 9　Execution time of program1.

大動作周波数を基に RTL シミュレーションを行う．設計ツールには，Xilinx ISE Design Suite 14.6，評価ツールには Xilinx ISE Simulator 14.6 を用いた．また，DalvikCore の実装デバイスには Spartan-6 XC6SLX45T を対象とした．

評価に用いるプログラムは次の 2 種類である．

(1) 1 から N までの自然数の和を求めるプログラム
(2) 要素 N の配列をソートするプログラム

（1）は 2 回の add-int を分岐命令によって N 回繰り返すコードである．（2）は aget や aput といった配列要素の読み込み/書き込みを行うバイトコードを用いてソートを行うコードである．配列オブジェクトの生成は通常 Dalvik VM に処理を移して行われるが，本稿ではデータメモリ上に配列データが既に存在するものとして評価を行う．したがって，バイトコードはすべてアクセラレータで実行可能であり，例外等も発生せず，MIPS モードへの遷移は発生しない．

5.2　結果・考察

2 種類のプログラムにおける実行時間を**図 9** および**図 10** に示す．動的 DRMT に基づく冗長なロード命令の削減を行った場合，実行時間はプログラム 1 において平均約 4.15 倍，プログラム 2 において約 1.90 倍高速化された．

ここで，プログラム中で削減されるロード命令について考える．プログラム 1 について，実行時間の多くを占めるループ部分には，加算バイトコード add-int が 2 個，条件分岐バイトコード if-lt が 1 個含まれる．これらのバイトコードからはそれぞれ MIPS のロード命令が 2 個生成される．したがって，ロード命令を必要とするバイトコード中の全ての Dalvik レジスタが DRMT にヒットした場合，1 ループ中で計 6 個のロード命令が削減されることとなる．また，プログラム全体では 6N 個のロード命令が削減される．

一方，プログラム 2 について，主要部分である 2 重ループの内側で 2 個のロード命令を生成するバイトコードが 7 個，外側には 3 個存在する．したがって，要素数を N 個とした場合，プログラム全体では $(2 \times 7 \times N + 2 \times 3) \times N$ 個のロード命令が削減されることとなる．

両プログラムにおいて，ロード命令削減機構は多くの冗長なロード命令を削減しているかのように思われるが，実際にはプログラム 1 とプログラム 2 において高速化率に大きな差が見られた．その原因として，プログラム 2 では aget 命令や aput 命令のように，生成する MIPS 命令数が多いバイトコードが頻繁に使用されていたことが考えられる．削減できるロード命令は全てのバイトコードに共通して 2 個であるため，ロード命令以外の削減できない MIPS 命令が多く生成されてしまうと，結果としてその削減効果は小さくなってしまう．

また，**図 11** に示すとおり，プログラムの実行時間に対して，データメモリのリード/ライト待ちに伴う DalvikCore のストール時間の割合が非常に大きいことがわかる．これは，外部 DDR2 メモリへのアクセスレイテンシが，DalvikCore

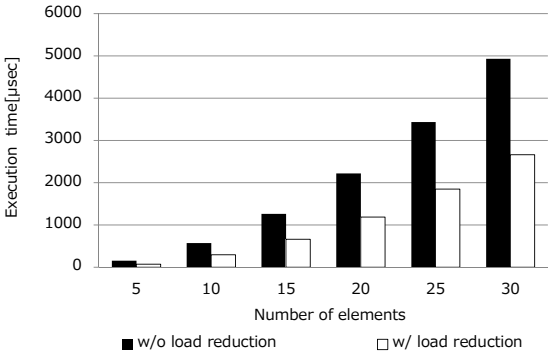

図 10　プログラム 2 の実行時間
Fig. 10　Execution time of program2.

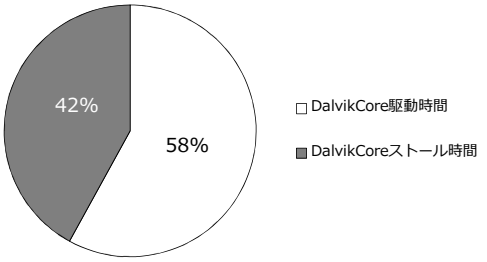

図 11　プログラム 1 における実行時間の内訳
Fig. 11　The ratio of execution time of program1.

表 2　DalvikCore の使用リソース数
Table 2　Resource usage of DalvikCore.

DRMT のマッピング方式	静的マッピング	動的マッピング
Slice 使用数	1,237（18%）	1,878（27%）
FF 使用数	1,846（3%）	2,124（3%）
LUT 使用数	3,994（14%）	5,603（20%）
最大動作周波数	39.522MHz	28.773MHz

の実行性能に大きな影響を与えていることを意味する．

5.3 使用 FPGA リソース数

DalvikCore の使用リソース数を**表 2** に示す．静的マッピングの DRMT および動的マッピングの DRMT では，使用 Slice 数に約 10%ほどの差があった．また，MipsCore の使用 Slice 数は 555 であり，Dalvik アクセラレータの接続によって使用リソース数が約 3.3 倍となった．

プロセッサコアに対するアクセラレータ部の使用リソース数が多くなったが，これは命令テーブルのサイズが大きく影響していると思われる．しかし，一般的なプロセッサはメモリレイテンシを隠蔽するため，プロセッサ内部にある程度のサイズのキャッシュメモリが設けられており，DalvikCore のようにキャッシュレスのプロセッサは非現実的である．例として，512KB キャッシュの搭載を想定する．単純な比較は難しいが，キャッシュのゲート数を約 280 万ゲートと見積もり [17]，FPGA の 1Slice を 10 ゲートと換算しても，キャッシュを含んだプロセッサに対してアクセラレータが及ぼすゲート増加量は 1%以下である．

6. まとめと今後の課題

本稿では，我々が提案する Dalvik アクセラレータについて，これまでの研究では行われていなかった，Dalvik レジスタ-MIPS レジスタ間の動的マッピングテーブルおよびこれに基づく冗長なロード命令削減機構を設けた．また，既存の MIPS ソフトコアプロセッサにアクセラレータを接続し，バイトコード実行可能なプロセッサ，DalvikCore としてバイトコード実行の高速化率を評価した．その結果，削減を行わない場合と比較して大幅に実行時間を短縮可能であることを示した．

今後の課題として，第 1 に DalvikCore の実行性能の低下を防ぐため，DalvikCore 内へのキャッシュメモリの搭載が必須と考える．

第 2 に，Android の DalvikCore へのポーティングに向けて，メモリ管理ユニット（MMU）やストレージ，I/O などを設ける必要がある．MMU をもたない組込み機器用プロセッサ向けの Linux も存在するが，少なくとも Android を動作させるには，複雑なアドレス変換機能やメモリ保護機能を持つメモリ管理ユニットが必要である．また，既に Android の動作実績のある [14]MIPS プロセッサシミュレータ，SimMips[16] も用いて研究を続ける予定である．

謝辞　本研究の一部は，文部科学省特別経費「持続可能社会にむけた知的情報空間技術の創出」および JSPS 科研費基盤研究 (C) 25330067 による支援を得た．ここに記して感謝する．

参考文献

[1] 株式会社カンター・ジャパン：2013 年 11 月から 2014 年 1 月のスマートフォン販売シェア調査，株式会社カンター・ジャパン（オンライン），入手先 ⟨http://kantar.jp/whatsnew/2014/03/03/NewsRelease_140304_ComTech.pdf⟩（参照 2014-03-04）．
[2] 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 (SACSIS) ポスター・セッション，Vol. 2010, No. 5, pp. 113–114 (2010)．
[3] 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 (ACS)，Vol. 4, No. 3, pp. 115–132 (2011)．
[4] Ohta, A., Yoshizane, D. and Nakajo, H.: Cost Reduction in Migrating Execution Modes in a Dalvik Accelerator, *Proc. 1st IEEE Global Conf.Consumer Electronics(GCCE)*, pp. 502–506 (2012)．
[5] 吉實大輔，太田淳，三輪忍，中條拓伯：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 (ESS)，pp. 225–226 (2012)．
[6] Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *Proc. Java Virtual Machine Research and Technology Symposium(JVM'01)* (2001)．
[7] Stoodley, M., Ma, K. and Lut, M.: Real-time Java, Part 2: Comparing compilation techniques, IBM (online), available from ⟨http://www.ibm.com/developerworks/jp/java/library/j-rtj2⟩ (accessed 2014-04-13)．
[8] McGhan, H. and O'Connor, M.: PicoJava: a direct execution engine for Java bytecode, *IEEE Computer*, Vol. 31, No. 10, pp. 22–30 (1998)．
[9] Silc, J., Robic, B. and Ungerer, T.: *Processor Architecture: From Dataflow to Superscalar and Beyond*, chapter 1.7.6, Springer (1999)．
[10] ARM: *ARM Architecture Reference Manual* (2005)．
[11] Porthouse, C.: Jazelle DBX Technology: ARM Acceleration Technology for the Java Platform (2005)．
[12] Steele, S.: Accelerating to Meet the Challenge of Embedded Java (2001)．
[13] Cheng, B. and Buzbee, B.: A JIT Compiler for Android's Dalvik VM, *Google I/O* (2010)．
[14] 太田淳：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，博士論文，東京農工大学大学院工学府電子情報工学専攻 (2013)．
[15] 吉實大輔：Android における中間コード直接実行機構によるハードウェア・アクセラレーション，修士論文，東京農工大学大学院工学府情報工学専攻 (2012)．
[16] 渡邉伸平，藤枝直輝，若杉祐太，高前田伸也，森洋介，吉瀬謙二：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 (開発支援・開発手法)，情報処理学会研究報告 2008-EMB-10，pp. 23–28 (2008)．
[17] 株式会社日本コンピュータ：P-RISC コア，株式会社日本コンピュータ（オンライン），入手先 ⟨http://www.nihoncomputer.co.jp/PRISC.htm⟩（参照 2014-04-13）．