Vol.2010-SLDM-144 No.18
Vol.2010-EMB-16 No.18
Vol.2010-MBL-53 No.18
Vol.2010-UBI-25 No.18
2010/3/26

Android プラットフォームにおける Dalvik バイトコードの CPU 負荷量の解析

間嶋 崇[†1] 横山哲郎[†2] 曾 剛[†1]
神山 剛[†3] 冨山宏之[†1] 高田広章[†1]

本稿では，ハードウェア独立な Dalvik バイトコードトレースから得られる情報を活用し，Android プラットフォームにおける CPU 負荷量の解析を行う．各 Dalvik バイトコードの CPU 負荷量を正確に精度良く解析するため，マイクロベンチマークの生成方法および実施方法を提示する．マイクロベンチマークの CPU 負荷量は，バイトコードの種類により最大 67 倍，引数のレジスタ値により最大 10 倍の差が存在した．したがって，アプリケーションの CPU 負荷量の正確なモデル化を行うためには，バイトコードの発行数のみならず，種類・引数を考慮する必要があることが明らかになった．以上により確立された解析手法を用いて，アプリケーション開発者に改善を提案するケーススタディを行った．実際に CPU 負荷量の削減方法を示唆できたことから，本解析手法の有効性が示された．

CPU Load Analysis Using Dalvik Bytecode on Android

Takashi Majima,[†1] Tetsuo Yokoyama,[†2]
Gang Zeng,[†1] Takeshi Kamiyama,[†3]
Hiroyuki Tomiyama[†1] and Hiroaki Takada[†1]

We present a CPU load analysis method on Android platform by using hardware-independent trace information of Dalvik bytecodes. For the purpose of analysis, methods for generating and executing micro-benchmarks which issue the sequence of each Dalvik bytecode are introduced. The experimental results showed that the CPU load of micro-benchmarks was largely affected by the types arguments of bytecodes. Specifically, the variation of CPU load was up to 67 times over the types and 10 times over the arguments of given bytecodes. Therefore, not only the number of issued bytecode but also the types of bytecode and given arguments should be considered to construct an accurate CPU load model. Through a case study, the effectiveness of our approach for reducing CPU load has been validated.

1. はじめに

CPU やメモリなどの使用量を解析・制御するリソース管理は，携帯端末のアプリケーション開発において特に重要である．ソフトウェア開発の早期においては，ターゲット端末が特定されておらず柔軟な設計が求められるため，シミュレーションや解析モデルを用いてホスト計算機上でリソースの見積りが行われる．通信量やメモリアクセス量は，ターゲット端末においても同じであり，また，ホスト計算機上でアプリケーションを実行することでこれらの量を取得できる．しかし，このリソース解析の方法では，CPU 負荷量を正確に見積ることは困難である．ホストとターゲットでも，また異なるターゲット同士でも，搭載されている CPU が違うためである．我々は，アプリケーションをターゲット携帯端末で実行した場合の CPU 負荷量を，ホスト計算機上で精度よく正確に見積る手法を開発することを目的とする．解析対象として，ハードウェアには Android[5] を搭載した携帯端末を選択し，ソフトウェアには Dalvik 仮想マシン（以下，DalvikVM）上で動作する Java アプリケーションをそれぞれ選択した．Java アプリケーションを選択したのは，ハードウェア独立な中間コードであるバイトコードを解析対象とし，端末に依存しない汎用の手法の確立を目指すためである．Android は開発環境のソースコードが入手可能であり，実際に仮想マシンを研究用に改良したり，複数メーカーの端末に容易にポーティングできるという利点がある．

本稿の貢献は以下の通りである．

- CPU 負荷量の解析に必要とされるマイクロベンチマークの作成法およびベンチマークの実施法を提示した．
- 実験を通して，CPU 負荷量の解析には，バイトコードの種類・引数・型など考慮する必要のある情報を明らかにした．
- アプリケーション開発者へどのような改善を具体的に提案できるかを明らかにし，ソフトウェア開発に本手法が貢献できることを示した．

2. 関連研究

Java 仮想マシン（以下，JVM）を対象としたハードウェア端末に非依存なプロファイリング手法は数多く提案されている．Binder らは，ハードウェア端末に依存しない指標であるバイトコード発行数，メソッド呼出し回数など限定的な情報のみを用いることで，ハードウェア端末から独立したプロファイリングを実現した[2]．JVM には，JVMPI（JVM Profiler

†1 名古屋大学 大学院情報科学研究科, Graduate School of Information Science, Nagoya University
†2 南山大学 情報理工学部, Faculty of Information Sciences and Engineering, Nanzan University
†3 株式会社 NTT ドコモ 先進技術研究所, Research Laboratories, NTT DOCOMO, Inc.

Vol.2010-SLDM-144 No.18
Vol.2010-EMB-16 No.18
Vol.2010-MBL-53 No.18
Vol.2010-UBI-25 No.18
2010/3/26

Interface）や JVMTI（JVM Tool Interface）などのプロファイリング用のインターフェースが用意されている．しかし，多くの組込み Java システムと同様に，DalvikVM 用にこういったインターフェースは確立していない．JProfiler[4] はエージェントをターゲット端末上で動作させ，ホストと通信を行ってプロファイルを行う．このプロファイリングエイジェントも上記のインターフェースを利用しているため，DalvikVM 上でそのまま使用することはできない．

文献 3) では，ホストで組込み Java システムのプロファイルを行うクロスプロファイラ CProf が提案された．実験が行われたすべてのメソッドキャッシュサイズとベンチマークの組み合わせにおいて，ハードウェア実装された仮想マシンにおける実行サイクル数の見積りの誤差は 3.3%以下であった．我々はソフトウェアによって実装された仮想マシンの CPU 負荷量の見積りを対象としているため，見積り誤差は文献 3) のものよりも大きくなることが予想される．

Albert らは，Java バイトコード列の計算量を再帰式の表現に落とし込み，機械的に解を求めるという静的解析のフレームワークを提案した[1]．彼らもプラットフォームに依存しないコスト計算を行っており，計算量の漸近的な振舞いを上手く扱うことができる．一方，我々は，実行されたトレースを用いて実際に掛かった CPU 負荷量を解析し，予測の絶対値の誤差を少なくすることを目指している．

Android において，Java はフロントエンドに使用されているだけであり，アプリケーションモデルは Java と全く異なる．実際，JVM はスタックマシンであるのに対し，DalvikVM はレジスタマシンであり，構造が異なり互換性がない．Dalvik バイトコードは Java バイトコードとは独立の言語であり，バイトコードが格納される DEX ファイルのフォーマットも class ファイルと異なる．我々の知る限りでは，本研究は Dalvik バイトコードを CPU 負荷量の解析に用いた最初の研究である．

3. CPU 負荷量解析モデル

本稿では，アプリケーション実行時の Dalvik バイトコードトレースに着目した CPU 負荷量解析モデルを提案する．ここでの CPU 負荷量とは，アプリケーション実行時の CPU 時間を指す．

本稿では，CPU 負荷量が実行された各バイトコードの CPU 負荷量の和で表されるモデルを仮定する．各バイトコードの CPU 負荷量は，非数値演算の場合はバイトコードの種類によって，数値演算の場合は種類のみならず引数のレジスタ値によって決定されるとする．本モデルへの入力は，解析を行う目的に適した性質を持つ．すなわち，バイトコードの種類および引数のレジスタ値は，ハードウェア端末に依存せず，データに再現性があり，取得が容易である．

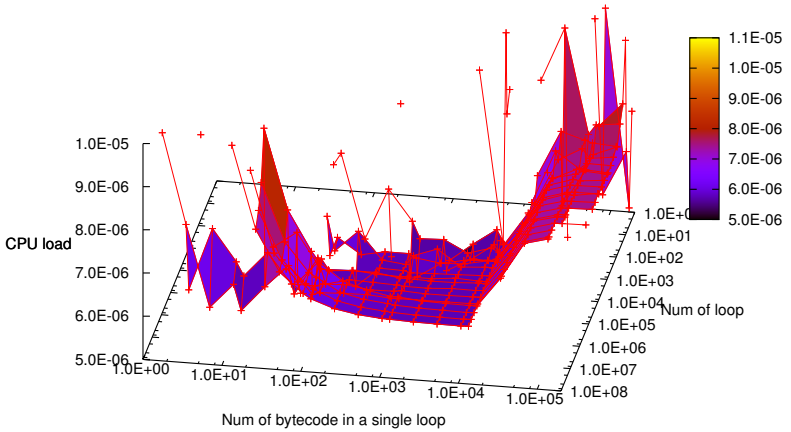


図 1 Android Dev Phone 1 におけるバイトコード 1 つあたりの CPU 負荷量
Fig. 1 CPU load for a single bytecode on Android Dev Phone 1.

各バイトコード 1 つあたりの CPU 負荷量を正確に求められること，バイトコード間の相互作用で CPU 負荷量が変化しないこと，数値演算の引数と CPU 負荷量の関係が簡易な式で表現できることが必要である．次章から詳細を見ていく．

4. 解析用ベンチマーク

4.1 マイクロベンチマーク

マイクロベンチマークの作成方法について説明する．各 Dalvik バイトコードの CPU 負荷量を正確に測定するため，各バイトコードが単一で連続発行された場合の CPU 負荷量のデータが必要となる．そこで，Dalvik バイトコードアセンブラ smali[8] によって，各バイトコードに対応するマイクロベンチマークを生成した．

マイクロベンチマークの生成にあたり，Dalvik バイトコードについて，ここで簡単に説明する．全 218 種類の命令は処理内容によって，**表 1** のように分類される．実測された CPU 負荷量の誤差・正確さを確認するため，マイクロベンチマークのループサイズとループ回数をスケールさせたときの CPU 負荷量を計測した．**図 1** は ADP1 において，ループサイズとループ回数をスケールさせた時の move 命令 1 回あたりの CPU 負荷量を示したものである．プロットに用いた負荷量は，各実測値から，空ループを対応する回数だけ実行したときの CPU 負荷量であるオフセットを差し引いた後に，ループサイズとループ回数で除算されて得られた値である．まず，ループサイズに着目すると，1000 以下のときは良

Vol.2010-SLDM-144 No.18
Vol.2010-EMB-16 No.18
Vol.2010-MBL-53 No.18
Vol.2010-UBI-25 No.18
2010/3/26

表 1 Dalvik バイトコードの概要
Table 1 Dalvik bytecode families

Opcode	Mnemonic	Purpose
00	nop	Waste cycles
01-0c	move*	Move between registers
0d,27	throw, ...	Exception handling
0e-11	return*	Return from a method
12-1c	const*	Constants to registers
1d-1e	monitor-*	Synchronization
1f-20	chech-cast, instance-of	Type checking
21,23-26	*array*	Array manipulation
22	new-instance	Instance creation
28-2c,32-3d	goto*,if-*, ...	Execution control
2d-31	cmp*	Comparations
44-51	aget*,aput*	Read/write array elements
52-5f	iget*,iput*	Read/write member fields
60-6d	sget*,sput*	Read/write static fields
6e-72,74-78	invoke*	Method invocation
7b-80,90-e2	add-*,mul-*,...	Operations
81-8f	*-to-*	Type conversion
73,79-7a,e3-ff		(unused)

い精度が得られなかった．一方，2000 以上の場合には良い精度が得られた．ループサイズが 10000 から 80000 に変化するときに，CPU 負荷量は急激に上昇した．これは命令キャッシュにマイクロベンチマークが乗り切らなかったことが原因であると考えられる．各バイトコードのサイズ 4B の 10000 倍である約 39.1KB は，ADP1 に搭載されている命令キャッシュサイズの容量 32KB を超えている．以上のことから，測定誤差の影響が少なくなるように，マイクロベンチマークのループサイズは 10000 に設定した．一方，ループサイズを 10000 に固定してループ回数を変化させたときループ回数が増加すると CPU 負荷量はほぼ単調に減少した．ループ回数が 10000 から 200000 の範囲では誤差は 3%以内であった．このため，ループ回数も 10000 と設定した．マイクロベンチマーク実行時の CPU 負荷量は，System.currentTimeMillis() によって得られる数値を使用した．これはミリ秒で測定した現在時刻と，協定世界時の UTC1970 年 1 月 1 日午前 0 時との実経過時間を返すメソッドであり，実世界の日付との対応のみに使われるべきである．しかし，本研究では，CPU に待ち時間のない場合を想定しているため，影響はないと考えられる．

4.2 評価用ベンチマーク

CPU 負荷量モデルの評価のため，以下の 3 種類のベンチマークを用いる． SciMark 2.0[7] は浮動小数点演算・最適化を対象としたベンチマークツールで，5 種類のテストプログラムを含む．Linpack はシステムの浮動小数点演算性能を計測するものである．これらはソースコードが公開されているため，プログラムの修正も可能である．Embedded CaffeineMark 3.0[6] は組込み機器用に開発されたベンチマークで，グラフィックのテストを行わない．6 種類のテストプログラムを含むが，ソースコードは公開されていないため，詳細な動作は確認できず，個々のテストプログラムを単体で実行できない．

本研究では，実験用に SciMark のソースコードを以下の点について修正した．まず一点目に，テストプログラムの演算結果のコマンドラインへの標準出力を消去した．これは，演算とは関係のない部分で CPU 負荷量を消費しないようにするためである．二点目に，5 種類のテストプログラムを連続で実行する仕様となっていたが，コマンドラインからの実行時に，テストプログラムを指定できるように変更した．この変更によって，各テストプログラム実行時の CPU 負荷量を測定できるようにした．三点目に，プログラムを実行する際に，コマンドラインからテストプログラムのループ回数を引数として手動で与えることができるようにした．なぜなら，SciMark のデフォルトの設定では，ベンチマーク実行にかかる負荷に応じて，プログラムが自動でループ回数を決定してしまう．したがって，この修正により，プラットフォームに非依存で再現性のあるベンチマークの実行が可能となる．Linpack はソースコードを修正せずに実行した．

5. CPU 負荷量解析

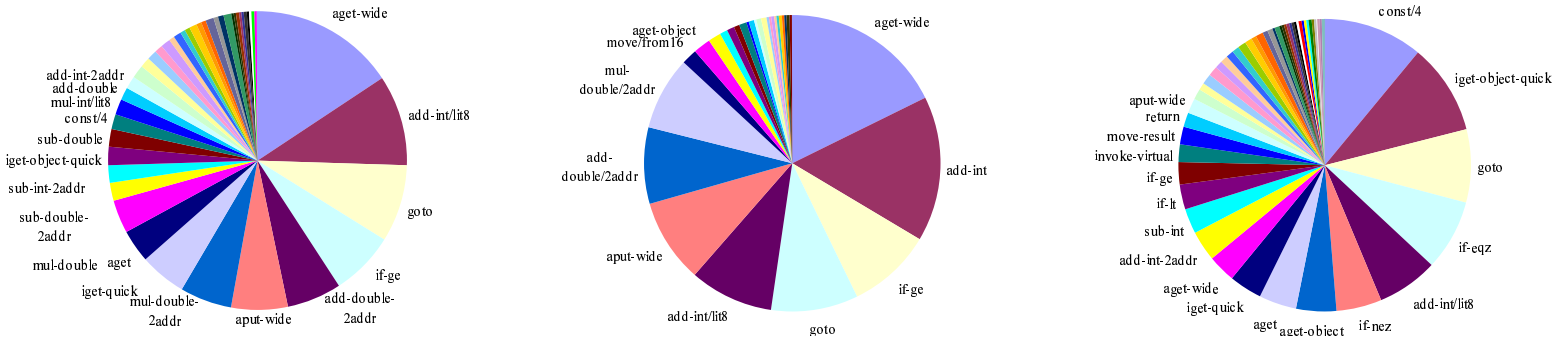
5.1 実 験 環 境

評価端末には，Android プラットフォームを搭載した HTC Android Dev Phone 1（以下 ADP1），Atmark Techno Armadillo-500 FX, Nokia N810 を用いた．**表 2** は評価端末の主な仕様を比較したものである．評価端末とホスト PC を USB ケーブルで接続し，Android SDK に付属する adb shell ツールにより，コマンドライン入力をバッチ処理で行うことですべての実験を行った．マイクロベンチマークを除いたベンチマークにおいて，CPU 負荷量は実機のパフォーマンスモニタユニットである/proc/stat の値を測定対象のベンチマーク実行の前後に取得し，その差分を算出した値を使用した．一方，バイトコードの実行トレースはホスト計算機上のエミュレータで取得した．取得に 100 倍以上のオーバヘッドがかかるため，CPU 負荷量とは別に実行し，取得する必要があった．なお，Java アプリケーションの実行トレースを取得できるよう仮想マシンを改造した．対象とするアプリケーションの実行以外の要素から CPU 負荷量へ影響を与えないようにするため，以下の点に留意し，実験を行った．動作モードをエアプレインモードに設定し，無線通信，音声をオフにした．また，ディスプレイ照明をオフに設定した．

5.2 ベンチマーク実行時のふるまい

CPU 負荷量解析モデルの形成において，バイトコードの情報の中で，CPU 負荷量への影響の大きい要素は何か調べたい．そこで，評価用ベンチマーク実行時のバイトコードのトレースを取得する予備実験を行った．**図 2** は各プラットフォームにおいて，各ベンチマークを実

Vol.2010-SLDM-144 No.18
Vol.2010-EMB-16 No.18
Vol.2010-MBL-53 No.18
Vol.2010-UBI-25 No.18
2010/3/26



(a) SciMark 2.0　(b) Linpack　(c) Embedded CaffeineMark 3.0

図 2　ベンチマーク実行時に発行された Dalvik バイトコードの比率
Fig. 2　Issued Dalvik bytecode percentages.

表 2　評価端末に搭載されるプロセッサ
Table 2　Processor specification of each platform

Target	Android Dev Phone 1	Armadillo-500 FX	N810
Processor	Qualcomm MSM7201A	Freescale i.MX31	TI OMAP 2420
CPU	ARM1136EJ-S	ARM1136JF-S	ARM1136
Clock	528 MHz	532 MHz	400 MHz
D / I cache	32 KB/32 KB	16 KB/16 KB	32 KB/32 KB
L2 cache	–	128 KB	–

行したときの，バイトコード発行回数の割合を示したものである．SciMark，Linpack，Embedded CaffeineMark のバイトコード総発行数はそれぞれ，251060274，4397488，2574642 であった．上位のバイトコードはバイトコード全種類の総数に対して限定的であり，SciMark のバイトコード総発行数が最も大きい値を示した．最小の CaffeineMark に比べ，10 倍ほどの差があった．続いて，**図 3** は各プラットフォームにおける，各ベンチマークの CPU 負荷量を比較したものである．ここで，JVM/x86 とは，Java バイトコードで構成されている各ベンチマークを，x86 上の JVM で実行した結果を表わしている．バイトコード総発行回数に対して，CPU 負荷量は比例の関係を持たないことが分かった．Linpack の方が CaffeineMark よりも，バイトコードの総発行回数は多いのにも関わらず，CPU 負荷量は CaffeineMark の方が大きかった．したがって，バイトコード総発行回数のみでは，CPU 負荷量を表すことができないことが示された．JVM を除くすべてのプラットフォームの SciMark の実行時間はほぼ等しかった．しかし，残りのベンチマークでは大きくばらついた．したがって，あるプラットフォームにおける 2 つのベンチマークの実行時間の比は，必ずしも別のプラットフォームにおけるそれに対応しないことが分った．すなわち，未知のプラットフォームにおけるあるベンチマークの実行時間を正確に見積るためには，バイトコードの総発行回数のみではなく，さらに詳細な解析が必要であることが判明した．ここで，ひとつ注意しなければならない．本実験では，高 CPU 負荷の場合を想定したため，ベンチマークは工学の分野で主に使用されるものを選んだ．したがって，add, mul といった数値演算系バイトコードの発行が大きな割合を占めているが，一般のアプリケーションでは関数呼び出し invoke など，異なる要素が含まれる可能性が高い．

CPU 負荷量は，実行の度に変動することが考えられる．CPU 負荷量が再現性を有するか確認するため，各ベンチマークを各ハードウェア端末で実行したときの変動係数について調査した．x86 上の JVM で実行した結果と，DalvikVM で実行した結果を比較した．すべてのベンチマークにおいて JVM は，DalvikVM に比べて高い変動係数を示した．すなわち DalvikVM 上のアプリケーションの実行時間のばらつきは JVM 上のものより小さかった．したがって，JVM のときよりも DalvikVM 上のアプリケーションの実行時間は精度良く測定できると考えられる．実機上の DalvikVM で実行した場合，ほとんどの観測値が平均から 5%以内に収まっていた．よって，CPU 負荷量は統計的にばらつきが小さく，再現性が

Vol.2010-SLDM-144 No.18
Vol.2010-EMB-16 No.18
Vol.2010-MBL-53 No.18
Vol.2010-UBI-25 No.18
2010/3/26

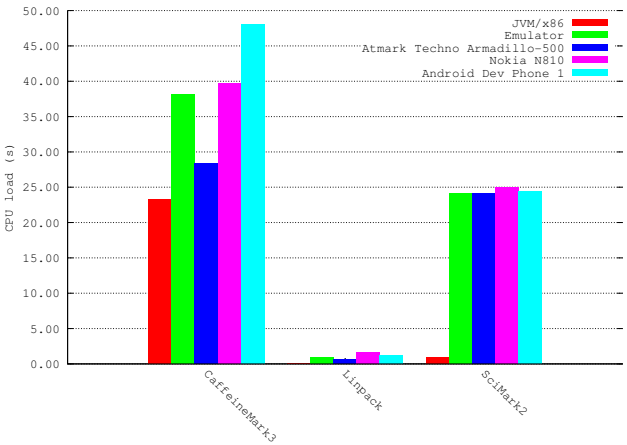


図 3 各ハードウェア端末における各ベンチマークの CPU 負荷量
Fig. 3 CPU load of each benchmark in each hardware platform.

あることが確認された．

5.3 バイトコード列が CPU 負荷量に与える影響

バイトコード発行回数から，モデルによって CPU 負荷を見積ることができるかについて，実験を行った．バイトコード発行が CPU 負荷量に対して線形性を持つか確認した．処理内容で分類されたバイトコードの各クラスから，1 つずつバイトコードを組み合わせて実験を行った．この実験では，比較対象の一方を div-int に固定し，バイトコードの組合せを作成した．2 つのバイトコードをそれぞれ繰り返し発行した場合の CPU 負荷の平均が， 2 つを交互に発行した場合と，ループ構造内で 5000 回ずつ発行した場合の CPU 負荷に合致した（**表 5.3**）．交互にバイトコードを発行した場合の CPU 負荷量と，各バイトコードの平均との誤差はわずか 5.0%以内であった．我々が観測した範囲では，バイトコード発行の前後関係に依存せず，CPU 負荷量に対して線形性を有することを確認した．このようにしてバイトコードの数を増やしていけば，一般のベンチマークの CPU 負荷量を精度よく見積ることができると考えられる．

5.4 各バイトコードの CPU 負荷量

各バイトコードの CPU 負荷量の特徴，および正確な解析モデル構築のために，どの程度バイトコードの種類を考慮すべきかを明らかにしたい．そこで，各 Dalvik バイトコードのマイクロベンチマークを実行したときの CPU 負荷量を測定した．各バイトコードに与える引数のレジスタや定数の値は無作為に選択したものを用い，実測は 30 回以上行った．

表 3 バイトコード発行の CPU 負荷量に対する線形性
Table 3 Bytecode linearity against CPU load

Bytecode	CPU load		
	Single	Alternate	5000/5000
div-int	22541	–	–
cmpl-double	29084	25866	25887
const	11471	16787	16927
div-double	260063	141689	141605
double-to-long	43137	32343	32231
goto	7402	14329	14384
if-eq	12499	17579	17535
move	5787	13523	13531
neg-int	6582	13933	13908
not-int	6582	13948	13908

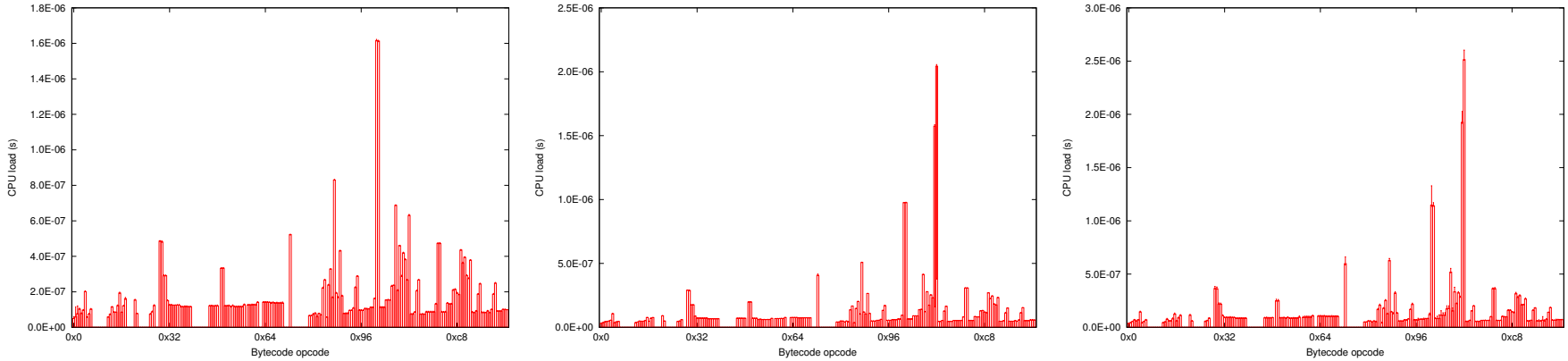
図 4 は各ハードウェア端末における，各バイトコード 1 回発行あたりの CPU 負荷量を示したものである．エラーバーは測定された値の最大，最小値を示したもので，横軸は Dalvik バイトコードのオペコード値を表す．各バイトコードの CPU 負荷量には大きな差が認められた．

バイトコードの種類によって CPU 負荷量は著しく異なった． 特に，ADP1 において，最大の CPU 負荷量を示した div-long(9e) の 1.62e-06s，および最小の CPU 負荷量を示した nop(00) の 4.97e-08s 間の比は約 32.5 倍にもなった．他の端末では，CPU 負荷量の最大値と最小値の比はさらに大きな値を示し，N810 では最大 67.3 倍であった．なお，CPU 負荷量が多かったバイトコード群は，long および double の除算・剰余・型変換（86, 8b, 9e-9f, ae-af），インスタンス変数への書込み・読込み（52-5f），静的メソッド呼出し（71）などであった．逆に CPU 負荷量が少なかった命令は nop（00），レジスタへの定数値の書込み（12-1b），分岐（32-3d），除算・剰余を除いた int・long・double の算術・論理演算（7b-80, 91-e2）などであった．

算術演算（91-e2）の間では，CPU 負荷量に大きな違いが認められた．一方，それ以外のバイトコードでは似た機能のものは，CPU 負荷量がほぼ同じ値を示したものが数多く存在した．特に，ゼロとの値の比較を行うブランチ（38-3d），静的フィールドの値の読み書き（63-6d），配列データの読み込み（47-4a）のバイトコード群の CPU 負荷量はそれぞれほぼ一定であった．

各バイトコードの CPU 負荷量は，ハードウェアに依らず共通する傾向と高い相関を示した．例えば，最大・最小の CPU 負荷量を示したバイトコードおよび同様な CPU 負荷量を示すバイトコードの集合は共通した．スピアマンおよびケンドールの順位相関係数は，.937 以上であった．

Vol.2010-SLDM-144 No.18
Vol.2010-EMB-16 No.18
Vol.2010-MBL-53 No.18
Vol.2010-UBI-25 No.18
2010/3/26



(a) Android Dev Phone 1　(b) Armadillo-500 FX　(c) N810

図 4　各ハードウェア端末における各バイトコードの CPU 負荷量
Fig. 4　CPU load of each bytecode in each hardware device.

しかし，それらのバイトコード群間の CPU 負荷量の比はハードウェア端末ごとに異なった．特徴的な所を 1 つ例示すると，他のバイトコードよりも著しく CPU 負荷量が大きかった div-long(9e) と rem-long(9f) は，この 2 者の比および全体の平均値からの比が端末ごとに大きく異なっていた．

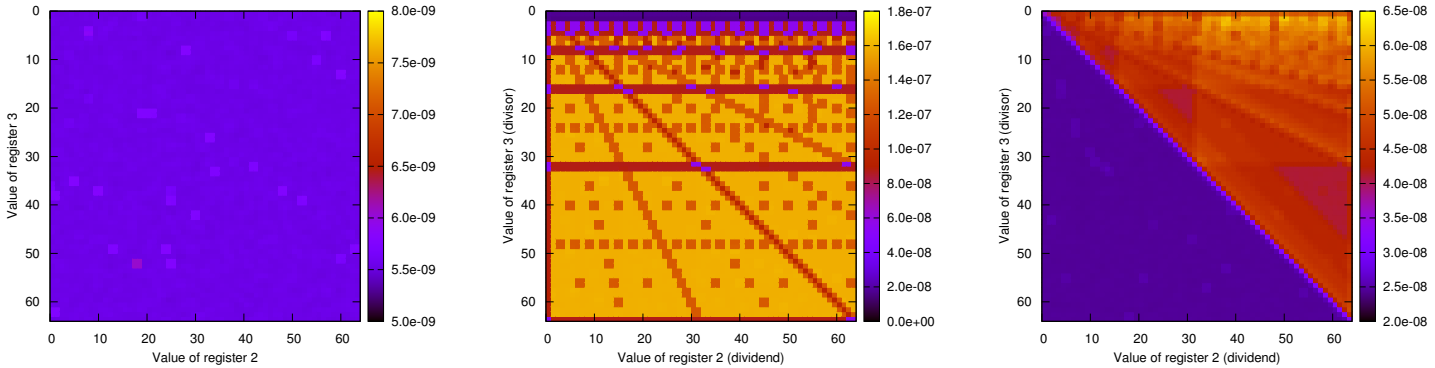
これらの結果から各バイトコードの CPU 負荷量の特徴および正確な解析モデル構築のために必要とされるものが明らかになった．CPU 負荷量の実測値のばらつきが小さいことは，解析モデルを用いてより高い精度の CPU 負荷量が得られる可能性を示している．CPU 負荷量は各バイトコードの種類に依存しておりその差は大きかったので，正確な CPU 負荷量を計算するためには，バイトコードの数のみならず，バイトコードの種類を考慮する必要がある．しかし，CPU 負荷量がほぼ同じであるバイトコード群はまとめて考えても正確さはそれほど損なわれないと考えられる．端末間で各バイトコードの CPU 負荷量は共通する傾向を有していたので，ある端末で作成した解析モデルの結果を定数倍することで，他の端末の結果をある程度予測することが可能である．しかし，より正確に CPU 負荷量を説明するためには，端末ごとに各バイトコードの CPU 負荷量を求めることで解析モデルを構築する必要がある．

5.5　引数の型・値による CPU 負荷量の変化

演算対象の値が，各バイトコードの CPU 負荷量にどれほどの影響があるのかを特定すること，およびその傾向を踏まえ，より正確な解析モデル化を実現する方法を明らかにするために実験を行った．例えば，バイトコード `div-double r1, r2, r3` は，double 型である `r2` および `r3` のソースレジスタの値を除算した結果を `r1` に格納している．この 2 つのソースレジスタの値に CPU 負荷量がどれほど影響され，どのようにモデル化されるかということを，本節で解明したい．

バイトコードへの引数を $[0, 64]$ の範囲の整数値で変化させたときの CPU 負荷量を，2 次元のグラデーションで示した結果，バイトコードの種類によってグラフの特色は大きく異なった．大きく分類すると，(a) ほぼ一定の CPU 負荷量が観測されたため，単色であるグラフ (cmp-*,add/sub/mul-int/long)，(b) 規則的に変化する CPU 負荷量が観測されたため，幾何的な模様が描かれたグラフ (*-float/double)，(c) 引数の大小によって異なる CPU 負荷量が観測され縦目，対角線の上下で色が異なるグラフ (cmp?-*,div/rem-int/long) の 3 種類であった．それぞれの分類の例を**図 5** に示した．すべての例は Armadillo-500 における実験結果である．分類 (a) では，add-int の実験結果を示した．2 つのソースレジスタの値が等しい場合に， CPU 負荷量が低い値を示した．次に，分類 (b) では，引数が変化する

Vol.2010-SLDM-144 No.18
Vol.2010-EMB-16 No.18
Vol.2010-MBL-53 No.18
Vol.2010-UBI-25 No.18
2010/3/26



(a) add-int (b) div-double (c) rem-double

図 5 バイトコードの引数による CPU 負荷量の変動
Fig. 5 CPU load variance caused by bytecode arguments.

と CPU 負荷量が規則的に変化した．これは図の色の規則的な変化から読み取ることができる．div-double の実験結果では，除数が 2 の累乗のときおよびに除数と被除数が同じ値の時に周囲よりも小さい CPU 負荷量であることが赤色の横線および対角線からそれぞれ読み取れる．分類 (c) では，rem-double の実験結果を示した．2 つのソースレジスタの保持する値の大小によって CPU 負荷量が異なるために，対角線の上下で色が異なっている．被除数が除数よりも大きいときに CPU 負荷量が大きくなった．ただし，除数が 2 の累乗のときは周囲よりも低い CPU 負荷量を示した．

レジスタ値の変化による CPU 負荷の変化量は，バイトコードの種類を変化させたときときと同様に，著しく大きな値が観測されることがあった．例えば，上記の div-double の実験結果では，最大および最小の CPU 負荷量の比は 10.4 倍にもなった．

引数のレジスタの値に依って 10 倍以上もの CPU 負荷量の差が観測されたことから，バイトコードの CPU 負荷量をより正確に説明するためには，引数のレジスタの値を考慮することが必要であると言わざるを得ない．しかし，引数の組み合わせすべてを実験によって試すことは現実的には不可能である．なぜなら，一例を挙げれば，倍精度の 2 数の組合せは $2^{64} \times 2^{64} (> 10^{38})$ 通りという膨大な量になるからである．したがって，現実の CPU 負荷量を上手く表現する軽量なモデル化が必要である．

軽量なモデル化は，グラフの特徴を踏まえ，各バイトコードの引数の分類と各分類中の要素が引数として組み合わせられたときの CPU 負荷量を求めておくことで実現できる．例えば，上記の `div-double` の実行時間は，除数が 2 の倍数のとき，除数・被除数のいずれかが 0 のとき，および引数が等しいときを除き，一定と見なした場合は，CPU 負荷量 $L_{\mathrm{div-double}}$ は以下のように表すことができる．

$$L_{\texttt{div-double}}(r_2, r_3) = \begin{cases} 16 & \text{if } r_3 \text{ is a multiple of 2} \\ 19 & \text{if } r_2 = 0 \text{ or } r_3 = 0 \\ 30 & \text{if } r_2 = r_3 \\ 160 & \text{otherwise} \end{cases}$$

ただし，r_2, r_3 は `div-double` に与えられた引数のソースレジスタ値を示す．

div 演算を行う時の数値型による CPU 負荷量の変動についても計測した（**図 6**）．double, long, float, int の順に CPU 負荷量は大きい値を示し， int, double それぞれの最大値の間には 5 倍ほどの差があった．long 型を使用した場合のみ，特異な CPU 負荷量の変動の傾向が観測された．CPU 負荷量が減少する引数の数が他の数値型の場合よりも少なく，最小値であっても 10000 を超す CPU 負荷量が観測され，最大値に対する変動の比率が小さかった．このように数値型によって CPU 負荷量が大きく異なる傾向は，アプリケーションを設計する際にも，定義する数値型によって動作に影響を及ぼすことを示す．

Vol.2010-SLDM-144 No.18
Vol.2010-EMB-16 No.18
Vol.2010-MBL-53 No.18
Vol.2010-UBI-25 No.18
2010/3/26

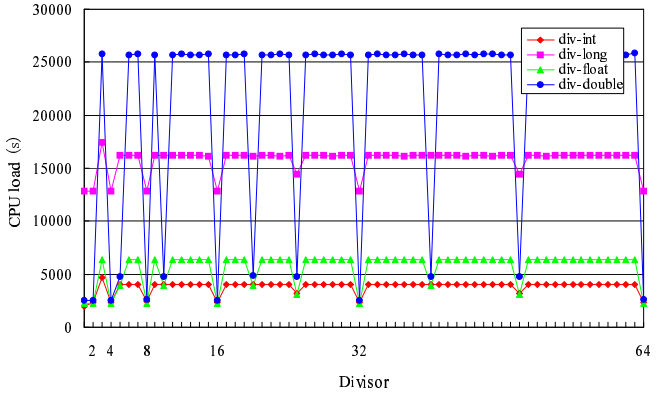


図 6 数値型による CPU 負荷量の変動
Fig. 6 CPU load variance caused by numeric types.

5.6 ユースケース

前節までの解析結果を用いて，アプリケーション開発者に，より効率的コードを教示できる事例を 2 つ示す．

まず，計算精度と CPU 負荷量のトレードオフを考慮し，採用するデータ型を決定するような大局的判断に活用できることを示す．例として，SciMark 中の double 型を float 型にすることで削減される CPU 負荷量を概算することを考える．SciMark 中の double 型に関する演算は約 25%であった（**図 2(a)**）．演算を double 型から float 型にしたときの CPU 負荷削減量は約 30-90%である（図 4(a)）．これらの数字を用いると，SciMark 中の double 型を float 型に置換することによる CPU 負荷削減量は約 8-25%であると概算できる．実測では CPU 負荷の削減量は約 15%であったので，アプリケーション開発者が判断を下す材料とする程度の正確さを有していると考えられる．

続いて，コードの一部が与えられたときに改善を教示するための局所的判断に活用できることを示す．long 型の変数 `x` に対し，丸め誤差の影響を除けば同一の計算を行う 2 式：

(1) 2 で除算後，double にキャスト：`(double)(x / 2)`

(2) double にキャスト後，2 で除算：`(double)x / 2`

の効率を比較するとする．どちらの式でもキャストに必要な CPU 負荷量は同じである．しかし，double 型より long 型の 2 除算の方が効率的であることが図 6 から読み取れる．したがって，(2) 式の方が効率的であると考えられる．5000 回の連続実行を 1 万回ループさせた実測では，CPU 負荷量の比は 2.85 倍であったので，以上の考え方の正しさを補強する結果が得られたと言える．

さらに，このような事例を応用すれば，アプリケーション開発者に対して，開発段階に携帯端末上で実行したときの消費エネルギーの値を提示すること，あるいは，CPU 負荷量の高いソースコードの箇所を検出し，省エネルギーなコーディングを教示することが可能となる．

6. おわりに

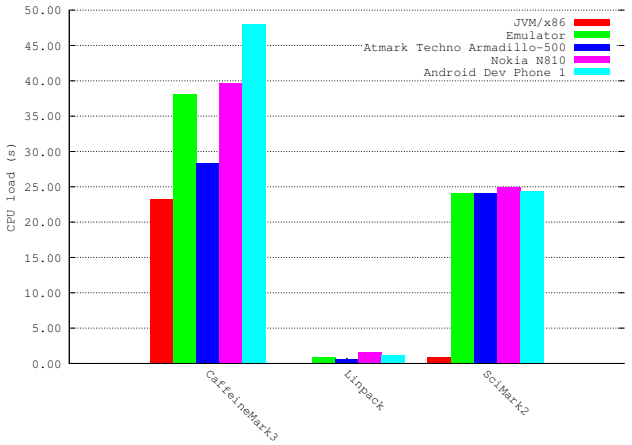
本稿では，ハードウェア独立な情報である Dalvik バイトコードトレースを活用し，Android プラットフォームにおける CPU 負荷量の解析を行った．評価端末には，Android Dev Phone 1, Armadillo-500 FX，N810 を使用した．まず，ハードウェア端末が与えられた時に，各バイトコード 1 つあたりの CPU 負荷量を正確に精度良く解析するためのマイクロベンチマークの作成方法と実施方法を提示した．次に，そのマイクロベンチマークによって CPU 負荷量への貢献要素を特定した．実験では，単独バイトコードの CPU 負荷量には，種類により最大 67 倍，引数のレジスタ値により最大 10 倍の差が確認された．したがって，アプリケーションの CPU 負荷量の正確なモデル化を行うためには，トレース中のバイトコード数のみでは不十分であり，バイトコードの種類や引数を考慮する必要があることが明らかになった．解析結果を応用することで，アプリケーションの設計者に CPU 負荷量を考慮した開発を提案できることが期待される．

参 考 文 献

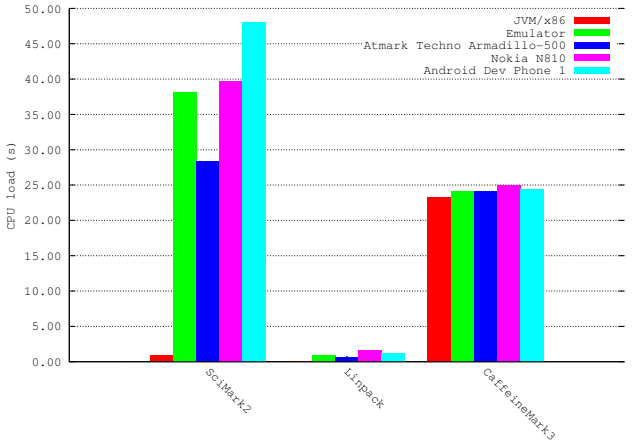
1) Albert, E., Arenas, P., Genaim, S., Puebla, G. and Zanardini, D.: Cost Analysis of Java Bytecode, *European Symposium on Programming*, pp.157–172 (2007).
2) Binder, W., Hulaas, J., Moret, P. and Villazon, A.: Platform-independent Profiling in a Virtual Execution Environment, *Software Pract. Exper.*, Vol.39, No.1, pp. 47–79 (2009).
3) Binder, W., Schoeberl, M., Moret, P. and Villazon, A.: Cross-profiling for Embedded Java Processors, *Proc. International Conference on Quantitative Evaluation of Systems*, pp.287–296 (2008).
4) ej–technologies: JProfiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.
5) Google Projects for Android. `http://code.google.com/android/`.
6) Pendragon Software Corporation: CaffeineMark 3.0. `http://www.benchmarkhq.ru/cm30/`.
7) SciMark 2.0. `http://math.nist.gov/scimark2/`.
8) smali. `http://code.google.com/p/smali/`.

【正誤表】

箇所	誤	正
4 頁，右段 1 行目	除く	含む
4 頁，右段 1 行目	SciMark	CaffeineMark
5 頁，図 3	右図参照	



(a) 誤



(b) 正

図 3 各ハードウェア端末における各ベンチマークの CPU 負荷量
Fig. 3 CPU load of each benchmark in each hardware platform.