フォアグラウンドアプリケーションのメソッドコール監視に基づく CPU 使用率推定とそれによるクロックレート制御

熊倉 孝太† 神山 剛†† 小口 正人††† 山口 実靖†

†工学院大学 電気・電子工学専攻 〒163-8677 東京都新宿区西新宿 1-24-2
††長崎大学 情報データ科学部 〒852-8521 長崎県長崎市文教町 1-14
†††お茶の水女子大学 理学部情報科学科 〒112-8610 東京都文京区大塚 2-1-1
E-mail: †cm22022@g.kogakuin.jp, ††kami@nagasaki-u.ac.jp
†††oguchi@is.ocha.ac.jp, †sane@cc.kogakuin.ac.jp

あらまし スマートフォンにおける最も大きな課題の一つはその大きな消費電力である．そのため，消費電力と端末性能に大きな影響を与える CPU クロックレートの適切な制御は重要なテーマである．しかし，端末の省電力化と性能向上の間にはトレードオフの関係性がある．CPU クロックレートを向上させると端末性能が向上するが消費電力量は増加する．逆に CPU クロックレートを低下させると端末性能は劣化するが消費電力量の削減がもたらされる．よって，端末の CPU リソースに対する過不足ないクロックレート制御が好ましい．しかし現在の Android スマートフォンでは，端末の CPU 使用率に追従したクロックレート制御を行っているため，CPU 負荷の状況とそれを追従するクロックレートに時間的なずれが生じ, 必ずしも適切な制御が行われてはいない. この課題に対して我々は過去の文献にて，フォアグラウンドアプリケーションのメソッド呼び出しに着目し，近い未来の CPU 使用率予測に基づいたクロックレート制御する考え方を提案した．本稿では，その実装の方法を提案し，我々の実装と Android OS に採用されている既存 governor と比較し有効性を示す.

キーワード 並列・分散処理, リアルタイム処理・リアクティブ処理, ストレージ管理

1. はじめに

Android OS はスマートフォン OS において全世界で 70%以上のシェア率を誇るプラットフォームである[1]. そのため，同プラットフォームの性能向上は重要なテーマである．また，消費電力量の大きさはスマートフォンにおける最も大きな課題の一つである[2]. スマートフォン内で多くの消費電力を消費する装置として CPU が挙げられる. CPU は端末の処理性能と消費電力量に大きな影響を与えるが，これらの間にはトレードオフの関係性が成り立つ. CPU クロックレートを上げることで端末の処理速度を向上させることができるが, 消費電力量が増加する. 反対に CPU クロックレートを下げることで端末の処理速度は低下するが，消費電力量の削減につながる．よって，性能低下を小さく抑えつつ，大きな消費電力の削減を実現することが好ましく，CPU 負荷が高いときのみ CPU クロックレートを増加させ，そうでないときに減少させることが重要であると言える.

Android OS は Linux ベースの OSS であり，カーネル部分には Linux カーネルが採用されている．また同カーネルは端末の CPU クロックレート制御を行う. その方法は端末の CPU 使用率の変化に追従して CPU クロックレートを変化させるものである．そのため同カーネルは CPU クロックレートの制御を行うが, 近い将来の端末内 CPU 使用率を予測することはない．よって，観察された過去の CPU 使用率と現時点や近い将来の CPU 使用率に大きな差がある場合には適切な CPU クロックレート制御が行われないことがある[3].

また，Android スマートフォンにおける CPU 使用率はフォアグラウンドアプリケーションの処理内容に大きな影響を受けることが分かっている[4]．よって，端末全体の CPU 使用率を予測するには, フォアグラウンドで実行されるアプリケーションが近い将来に使用する CPU 使用率を推測することが効果的であると考えられる．我々は，アプリケーションが使用する CPU リソースは，アプリケーション内メソッドコールと大きな関係性があると予想しており，アプリケーション内メソッドコール監視に基づき，近い未来のアプリケーションによる CPU 消費量と端末全体の CPU 使用率を推定できると考えている.

本稿では，まず Android スマートフォンにおける既存の CPU クロックレート制御手法を紹介する. そしてフォアグラウンドアプリケーションのメソッドコール監視に基づく CPU 使用率推定手法と, それによる CPU クロックレートの制御手法を提案する．次に，関連研究としてフォアグラウンドアプリケーション内メソッドの観察手法と，CPU クロックレート制御に関する先行研究を紹介する．そして，この制御の実装方法の提案と, 自作アプリケーションを用いた本 CPU クロックレート制御の性能評価を行い，本手法の有効性や課題

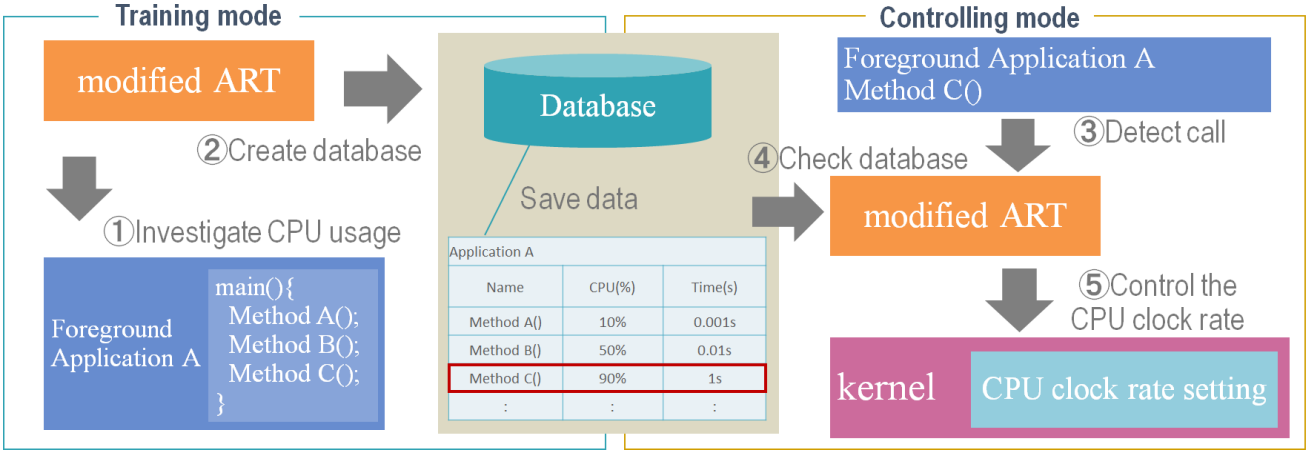

図 1　フォアグラウンドアプリケーションのメソッドコールに基づくクロックレート制御

について述べる．

2. FG アプリケーションのメソッドコール監視に基づくクロックレート制御

1 章で述べたように，Android スマートフォンにおいて，Linux カーネルは端末全体の CPU 使用率を取得し，その結果に基づいた CPU クロックレート制御を行う．近い将来の CPU 使用率を予測した制御は行わず，過去の CPU 使用率に追従した制御を行う．よって，観察結果と現在の CPU 使用率に相違がある場合には最適なクロックレートが選択されないことがある[4]．この問題点に対して我々は過去の研究[5][6]にて，Android スマートフォンにおけるフォアグラウンドアプリケーション内メソッドコール監視に基づき，近い未来の CPU 使用率を推定し CPU クロックレート制御を行う手法を提案した．本稿では提案手法を自作アプリケーションに適用し，その有効性を示しつつ課題点を考察する．

提案手法の概要を図 1 に示す．提案手法はフォアグラウンドアプリケーション内のメソッドコールを観察する Training mode と，観察結果に基づく CPU クロックレート制御を行う Controlling mode から構成される．Training mode ではフォアグラウンドアプリケーション内で実行されるメソッドごとの CPU 使用率を取得する（図中①）．また取得したメソッドごと CPU 使用率から，実行時間が長く CPU 使用率が極端に大きい（小さい）メソッドを特定する（図中②）．Controlling mode では Training mode で取得したメソッド情報を基に CPU クロックレート制御を行う．まず，フォアグラウンドアプリケーション内で実行されるメソッドを監視する（図中③）．次に，CPU 使用率が大きい（小さい）と予測されるメソッドが実行された際には（図中④），クロックレート制御を行い，CPU 使用率を上げる・下げるといった処理を行う（図中⑤）．

Android OS 内における既存の CPU クロックレート制御方法について説明する．同 OS に採用される Linux カーネルには CPU governor という機能がある．governor は端末の CPU 使用率に対して動的に CPU クロックレートを変更している．選択される governor ごとに CPU クロックレートの変更ポリシーは違うため，シチュエーションに応じて使い分ける必要がある．以下で，実験端末の OS である Android 12 に実装されている governor について紹介する．

本稿で使用した Android 12 内には 4 種類の”schedutil”, “powersave”, “performance”, “userspace”という governor が採用されている．schedutil govoernor はスケジューラが取得した CPU 使用率に基づいて CPU クロックレートを変更する．また端末内で通常起動している governor は schedutil である．powersave governor は端末の消費電力削減を重視した governor である．そのため積極的に CPU クロックレートを下げるように動作するが，処理性能が低下するといった特徴がある．performance governor は処理性能向上を重視した governor である．そのため積極的に CPU クロックレートを上げるように動作するが，消費電力量が増加するといった特徴がある．userspace governor は CPU クロックレートをユーザが設定可能な governor である．本稿では既存の schedutil, powersave，そして提案手法を適用した schedutil の 3 パターンにおいて処理性能と消費電力量を比較し，評価する．

3. 関連研究

3.1. Android アプリケーションのメソッドコールに基づいたクロック周波数制御に関する研究

我々は過去に，フォアグラウンドアプリケーションのメソッドコール監視に基づく CPU 使用率推定方法を提案し，それによる CPU クロックレートを制御する考え方(コンセプト)を紹介した[5]．

これら研究では，JVM(Java virtual machine)環境を改

変し，フォアグラウンドアプリケーション内で実行されるメソッドごとに CPU 使用率を推定した. メソッドコールごとにその時点における累積 CPU 使用時間を取得し，メソッドの実行時間に対するメソッドコールとリターン間の累積 CPU 使用時間の割合をメソッドごとの CPU 使用率として求めた.

また, 我々は上記に紹介した CPU クロックレート制御手法を用いたアプリケーション内実行メソッドの観察と CPU 使用率の推定に関する調査を行った[6][7][8]. 自作アプリケーションを用いたメソッド観察においては，CPU 使用率が高いメソッドと低いメソッドを実行した際に CPU 使用率が 100%に近い状態と非常に低い(0.1%ほどしかない)状態を記録した. それにより, 我々の提案するフォアグラウンドアプリケーション内のメソッドコール観察手法は正しくメソッドごとの CPU 使用率を判別可能であると示した．また Google Play Store 内の実アプリケーションを用いたメソッド観察を行い，実行されるメソッドごとの所要する時間と CPU 使用率を観察できることを示した.

3.2. Android スマートフォンにおけるクロックレート制御に関する研究

文献[4]では，Android スマートフォンにおけるアプリケーション動作に着目し，性能を大きく劣化させることなく消費電力量削減することを実現している．具体的には, クロックレートを上げる際にはより小さく，下げる際にはより大きく変更する様に改変を行っている．ただし，性能評価はゲームアプリケーション 1 個のみに対するものである．そのため様々な種類のアプリケーション負荷を計測し，アプリケーションごとに動的な周波数制御を行うことで，より効果的な制御が行える可能性を示唆している.

先行研究[9]ではアプリケーションからの性能フィードバックを受け取ることで動的な CPU クロックレート制御を行っている．提案されたフレームワークでは 1 秒ごとに達成フレームレートを算出し，レートが規定値を上回る場合と下回る場合のそれぞれで CPU クロックレート制御を行っている.

また先行研究[10][11]において，Android スマートフォンにおける起動性能や，アプリケーション開始時に実行されるメソッドに対する詳細な分析が行われている．実行されるアプリケーションごとにメソッドの処理時間には差があり，特定のメソッドが多くの処理時間を要していることを示している.

3.3. Android スマートフォンにおける governor 動作を考慮した CPU クロックレート制御に関する研究

既存の governor 性能の問題点を指摘しつつ，アプリケーション動作に着目し最適なシステム構成（CPU クロックレートとメモリ帯域幅）を動的に選択する研究[12]が存在する. 本稿における提案手法と同様に, オフラインプロファイリングとオンライン制御の 2 段階の制御理論から構成されており，6 つの実アプリケーションにおいて 4~31%ほどの省電力化を達成している.

また Bao らはコンパイル時に CPU クロックレートを選択する手法を提案している[13]．既存の動的な CPU クロックレート制御の処理時間等の問題点を示しつつ，アプリケーション実行時の監視を行うことなく powersave governor を上回る消費電力量削減と処理性能向上を達成している.

フォアグラウンドアプリケーションの動作を考慮したクロックレート制御手法[14]がある．当該研究では, 自作ベンチマークと userspace governor を用いてフォアグラウンドアプリケーションの動作を追跡しつつ CPU クロックレート制御を行うことで，端末性能を保ちつつ消費電力量を削減できることを示している．しかし,提案された手法ではアプリケーション内のメソッドまでは考慮されていない.

4. フォアグラウンドアプリケーションにおける各メソッドの実行時間と CPU 使用率

文献[8]におけるフォアグラウンドアプリケーション内メソッドごと，CPU 使用率の推定結果を紹介する. 測定対象としたアプリケーションは Google Play Store 内で無料ダウンロードランキング上位 15 個のアプリケーションである（2022 年 4 月 28 日時点）. これら 15 個のアプリケーションは，メッセージアプリ 4 個，カメラアプリ 3 個，モバイルオーダーアプリ 1 個，ビデオ会議アプリ 1 個から構成されている．またこれらをアプリケーション A~O と呼称する．実験端末は Pixel XL であり，CPU コア数は 4 つである.

アプリケーション A~O において，最も CPU 使用率が高く実行時間が長いメソッドが多かったアプリケーション O における，CPU 使用率（横軸）とメソッド実行時間（左縦軸）の関係性を図 2 に示す．CPU 使用率はメソッド開始から終了までの累積 CPU 使用時間を経過した実時間で割った比率である. 端末内 CPU コア数が 4 つであるため，最大の CPU 使用率が 4.0(400%)付近の値を取った．図中の各プロットはメソッドコールを表している．図中の灰色の線は全メソッドコールにおけるスカイライン表示[15]を表している．スカイラインにより各メソッド実行時間の最大 CPU 使用率を直線で結び，測定アプリケーション内の全てのメソッドがスカイラインの左側に存在することを示した．オレンジ色の線は CPU 使用率ごとにグループ化したメソッドコールの頻度（右縦軸）を表している．CPU 使用率が 1.0 と 4.0(100%と 400%)付近にメソッドが集中していることより，このアプリケーションでは CPU コアを 1 つもしくは 4 つ使用するメソッドが多いこと

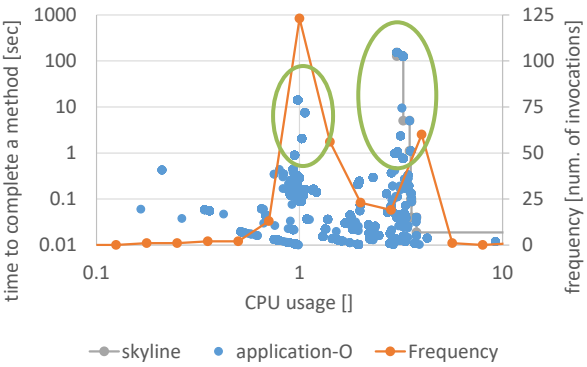

図 2　メソッドコールごと CPU 使用率と実行時間(Application O)

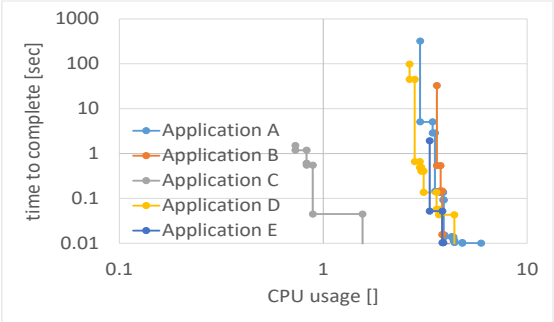

図 3　CPU 使用率と実行時間に伴うスカイライン表示(Application A~E)

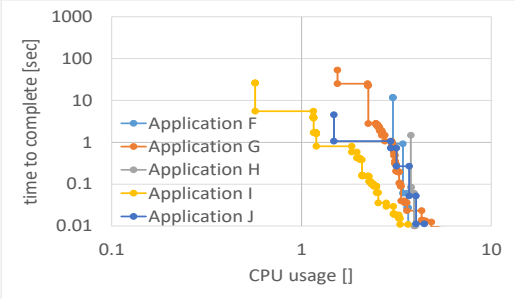

図 4　CPU 使用率と実行時間に伴うスカイライン表示(Application F~J)

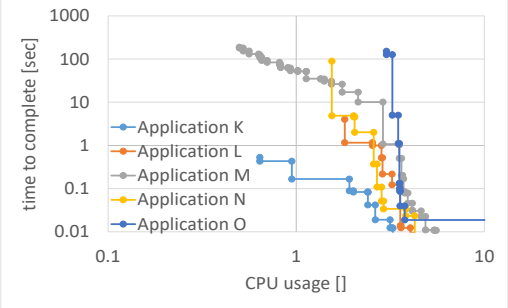

図 5　CPU 使用率と実行時間に伴うスカイライン表示(Application K~O)

が推測される．また本グラフでは実行時間が 10 ミリ秒以上のメソッドのみを表示している．次に，図からは一部のメソッドにおいて実行時間と CPU 使用率共に大きな値を取っていることがわかる．そのためこれらメソッドの呼び出しを検出したタイミングで CPU クロックレート制御を行うことが効果的であると考えられる．図中緑色のサークルに着目すると，実行時間が 1 秒以上のメソッドは 100%もしくは 400%近い CPU 使用率を記録していたことがわかる．つまりこれらメソッドのように多くの処理時間を要するメソッドは同様に多くの CPU リソースを必要としていると考えられる．

図 3～5 は全 15 個のアプリケーション A~O の測定結果をスカイライン表示にてまとめたものである．この結果から，アプリケーション O 以外にも実行時間が長く CPU 使用率の高いメソッドを持つアプリケーションが存在することが分かる．また一部のアプリケーションでは CPU 使用率は低いが，実行時間が 10 秒ほどのメソッドも存在していた．実行時間が長く CPU 使用率が高い・低いメソッドコールに対し提案手法を適用することで，近い将来の端末 CPU 使用率を予測したクロックレート制御が可能であると考える．

5. クロックレート制御の実装手法の提案

本章にて，我々が文献[5]にてコンセプトを提案したクロックレート制御手法の実装方法を提案する．我々が提案するシステムは ART(Android run time)環境の改変と，OS 内 Linux カーネル部分の改変により構成される．Training mode の実装方法は先行研究[6]に提案，紹介されており，本稿では Controlling mode の実装方法に焦点を当てて述べる．

まず Android OS の改変内容として，アプリケーション実行を司る ART(Java インタプリタ)の機能を改変する．インタプリタが実行アプリケーションのメソッドコールを処理する際に，CPU クロックレートを制御(クロックレートを設定)できる様に ART 実装を改変する．

CPU クロックレートの制御は，Android OS のカーネル(Linux カーネル)内の関数を呼び出すことにより可能であり，同カーネルを改変し一般ユーザ権限でも改変ができる様にする．具体的には，読み込まれると CPU クロック周波数を変更する procfs 内のファイルを作成し，ART よりこのファイルを読み込むことによりクロックレートを制御する．カーネル内のクロックレートを設定する qcom-cpufreq.c ファイル内の set_cpufreq()関数を改変し，procfs の読み込みが行われた際には，クロックレートを最高や最低に設定する機能を追加する．

本稿では，カーネルの CPU クロックレート自動制御機能の実装 schedutil に加えて，本システムによる制御

Application

```
class Thread1 extends Thread {
        double pl;
        public void run() {
                for (int i = 0; i < 10000; i++) {
                        for (int j = 0; j < 5000; j++) {
                        pl = pl * 1.01;
                        pl = pl * 0.09;
                        }
                }
        }
}
Method A(){
  final int nThread = 100;
        Thread1[] ths = new Thread1[nThread];
        for(int i=0; i<nThread; i++){
           ths[i] = new Thread1();
           ths[i].start();
        }
        try{
           for(int i=0; i<nThread; i++){
              ths[i].join();
           }
        } catch (InterruptedException e) {
           System.out.println(e);
        }
}
Method B(){
        try {
                Thread.sleep(10000);
                } catch (InterruptedException e) {
        }
}
```

図 6　自作アプリケーション内実行メソッド

も行われる実装を想定しており，本システムが制御を行ったあとに schedutil governor がクロックレートを上書き設定すると，その設定が採用されることになる．よって，CPU クロックレート制御の影響や効果は限定的であるとともに，現在の governor 実装に近い安全な動作が得られることとなる．

6. 性能評価

自作アプリケーションに対し提案手法を適用し，その動作の検証と性能の評価をした．使用した自作アプリケーションの概要を図 6 に示す．アプリケーション内には CPU 負荷の大きい method A と CPU 負荷の小さい method B が存在する．まず method A はスレッドを 100 個作成し，それぞれが 10000*5000 回の for 文を実行するメソッドである．これらスレッドは同時に実行されるため，端末内の CPU 全てに負荷を与え，CPU 使用率を 100%に近づける．次に method B は 10 秒間 sleep を行う．アプリケーション内のボタンをタップすると動作を開始し，これら method A・B を交互に計 100 回繰り返す．この処理に必要とした時間と消費電力量を測定する．

実験環境の使用端末は Pixel3a，CPU コア数は 8，

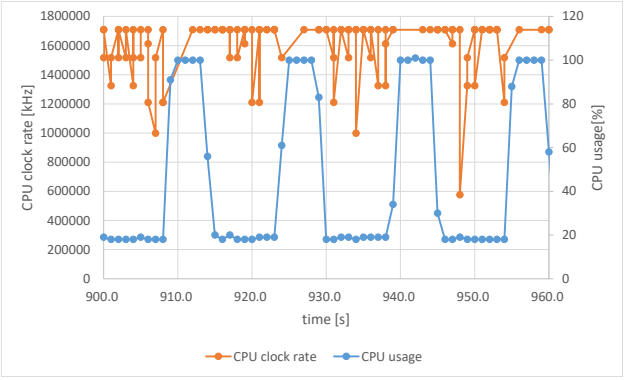

図 7　schedutil における
自作アプリケーションの実行結果

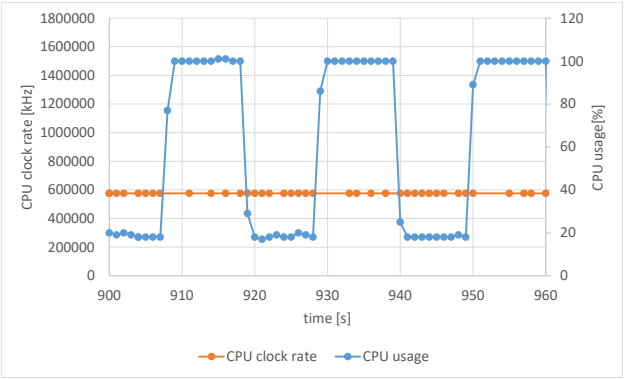

図 8　powersave における
自作アプリケーションの実行結果

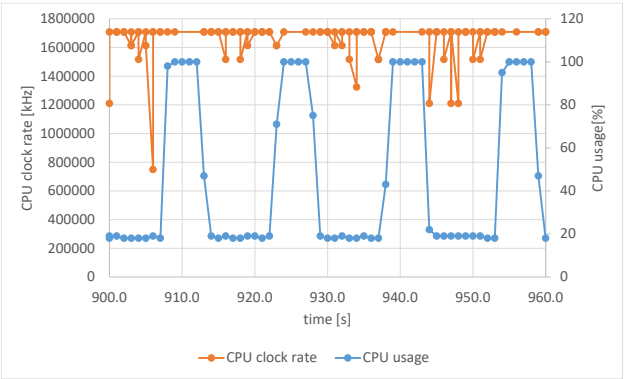

図 9　提案手法を適用した schedutil における
自作アプリケーションの実行結果

Android OS のバージョンは 12，使用したアプリケーションは上記の自作ベンチマークである．

提案手法による CPU クロックレート制御を適用していない schedutil，提案手法を適用していない powersave，提案手法を適用している schedutil において測定を行った結果を図 7~9 に示す．横軸はアプリケーション実行時の経過時間であり，それぞれの測定開始から 900 秒後の 60 秒間を抽出している．左縦軸は

CPU クロックレートを表しており，オレンジ色のライン上プロットが各時間における CPU クロックレートである．この値は procfs である scaling_cur_freq を参照することで取得した．右縦軸と青色のラインは端末全体の CPU 使用率を表している．CPU 使用率は vmstat コマンドから取得したユーザ空間とカーネル空間における CPU 使用率を足した値となっている．

図 7 に着目すると，schedutil はなるべく高い状態の CPU クロックレートを維持しようとする特徴があることがわかる．CPU 使用率が大きい状態では高い CPU 使用率を取っていたが，CPU 使用率が下がるとクロックレートを上げる・下げるといった処理を繰り返すことが分かる．

図 8 に着目すると，powersave は低いクロックレートを適用し続けることが分かる．CPU 使用率の上下によらず CPU クロックレートが低いままであった．そのため，CPU 使用率が上がった際のメソッド処理時間が図 7 の結果と比べると倍近く長くなっていることが読み取れる．

図 9 に着目すると，提案手法を適用した schedutil は図 7 の無改変な schedutil に近い動きを取ることが分かる．ただし，図 9 では CPU 負荷が高い method A の開始時に(すなわち CPU 使用率が上がる前に)CPU クロックレートを上げており，method A 中に CPU クロックレートが低くなる状況は回避できていることが分かる．ただし，頻繁に schedutil が CPU クロックレートを上書きするため動作に大きな違いは見られていない．

図 10,11 にそれぞれのアプリケーション実行と CPU クロックレート制御における処理時間と消費電力量を示す．それぞれの値は同環境の実験を 5 回行い，平均値を取得したものとなっている．消費電力量は Android 端末内 setting アプリケーションを改変し，アプリケーションごとに所要する消費電力量の推定値(ユーザ ID ごとに setting 内で計算された値)をログ出力した値となっている．

図 10 はそれぞれの測定におけるアプリケーション実行に所要した時間（CPU 負荷の重い・小さいメソッドを交互に 100 回繰り返すことに必要とした時間）を表している．最も処理時間の長かった手法は CPU クロックレート制御を行わない powersave であった．二つの schedutil は，ほぼ同等の処理時間を要しており，提案手法を用いたものが僅かに(本測定の例では約 0.004%)長くなっている．

図 11 の消費電力に着目すると，最も大きな消費電力量を必要としたのは powersave であることが分かる．powersave は消費電力の削減を目指す governor であるが，CPU クロックレートを下げ，最も長い処理時間を必要としていたため，結果として大きな消費電力量を

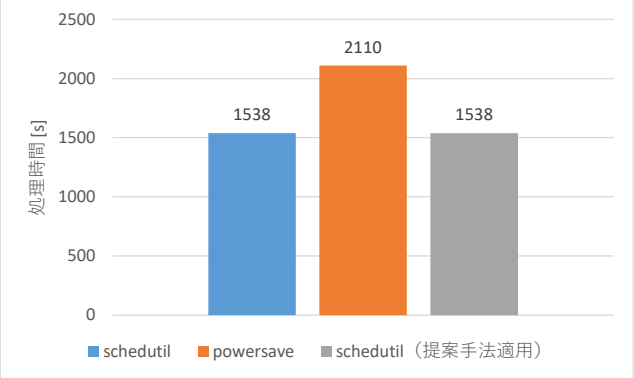

図 10　提案手法と既存 governor における処理時間の比較

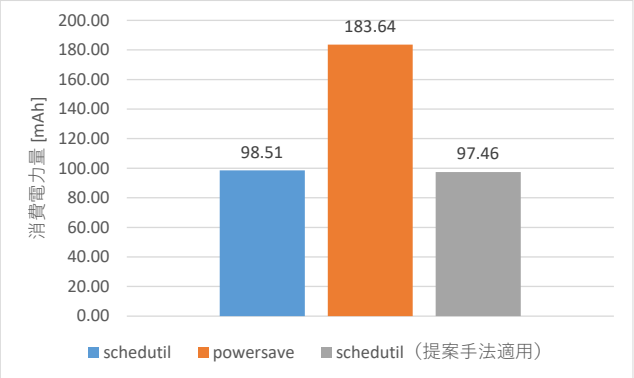

図 11　提案手法と既存 governor における消費電力量の比較

消費した．提案手法を用いた schedutil と，用いていないものを比較すると，提案手法により約 1mAh(約 1.1%)ほど提案手法が消費電力量を削減できたことが分かる．この結果は，schedutil の特徴である追従型の制御に対し，メソッドコールに対応した CPU クロックレート制御を追加した結果であると考えられる．以上より，提案手法により非常に小さな性能劣化で消費電力の削減が実現できたと言える．

7. 考察

図 7,9 より，schedutil は頻繁に CPU クロックレートを変更するように動作していることがわかる．この理由として，schedutil 内の sugov_update_shared()というメソッドが頻繁に実行されていることが原因である．schedutil の実行時には上記メソッドが毎回実行されており，内部では次の周波数を設定する next_f 変数を頻繁に更新している．CPU クロックレートの更新時に電力を消費しているなら，sugov_update_shared()を制御することで消費電力の少ない CPU クロックレート制御が達成できると考えられる．そのためには，CPU クロックレートの変更にともなう電量消費量の調査が必要となる．

また，クロックレート制御自体に必要とする時間とメソッドの実行時間との関係性を考慮することが重要であると考える．文献[8]では実アプリケーション内でCPU 負荷の大きなメソッドの実行に必要とした時間は約 0.1 秒以上であった．今回の結果よりクロックレート制御自体に必要とする時間を計算すると平均して0.002 秒程度であった．そのため，我々の提案するクロックレート制御はメソッドコール時間よりも短い時間で実現可能だと考えられる．また本稿ではCPU 負荷の高い・低いメソッドのどちらも 10 秒ほどのメソッド実行に対しての制御を行ったが，実アプリケーションにおいてはより実行時間が短く，CPU 負荷が自作アプリケーションほど大きな値を取らない可能性が高い．その際には今回のような最大・最小といった簡易的なCPU クロックレート制御ではなく，より細かい値を設定することやその頻度等も考慮する必要がある．また，CPU クロックレートを 1 回更新する際に必要となる処理時間と同様に，消費電力のオーバーヘッドを計測することも重要であると考えられる．

また文献[8]より，メソッドの CPU 負荷は CPU コアをいくつ使用しているかといった影響を受ける可能性がある．そのため CPU クロックレート制御を CPU コアごとに考慮することで，アプリケーションごとやメソッドごとにより最適な CPU クロックレート制御が行える可能性がある．

8. おわりに

本稿では，Android スマートフォンにおけるフォアグラウンドアプリケーションのメソッドコール監視に基づく近い未来の CPU 使用率推定とそれによる CPU クロックレート制御の考え方を紹介した．そして，その実装方法を提案し，提案実装の性能を自作アプリケーションの実行から評価した．その結果本手法は既存 governor の処理性能をほとんど損なうことなく，小さな消費電力量の削減を実現できることを示した．また本手法におけるクロックレート制御のオーバーヘッドを考慮しつつ，実アプリケーションに適用する際の懸念点や課題を考察した．

今後は，実アプリケーションに対してのメソッドコールに基づく CPU 使用率の推定とクロックレート制御を適用し，性能評価を行う予定である．

謝辞 本研究は JSPS 科研費 21K11854, 21K11874 の助成を受けたものである．

参 考 文 献

[1] Mobile Operating System Market Share Worldwide, Dec 2021 - Dec 2022. https://gs.statcounter.com/os-market-share/mobile/worldwide.

[2] P. K. D. Pramanik et al., "Power Consumption Analysis, Measurement, Management, and Issues: A State-of-the-Art Review of Smartphone Battery and Energy Usage," in IEEE Access, vol. 7, pp. 182113-182172, 2019, doi: 10.1109/ACCESS.2019.2958684.

[3] 関屋拓司，栗原駿，福田翔貴，濱中真太郎，小口正人，山口実靖, "アプリケーションの動作と消費電力を考慮したスマートフォン CPU クロック周波数制御", 情報処理学会 第 79 回全国大会講演論文集, 2017 巻, 1 号, pp. 127-128, 2017.

[4] Y. Sato, M. Oguchi and S. Yamaguchi, "Mobile Application Aware Smartphone CPU Clock Frequency Optimization," in 2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW), Takayama, Japan, 2018 pp. 564-566. doi: 10.1109/CANDARW.2018.00112

[5] K. Kumakura, A. Sonoyama, T. Kamiyama, M. Oguchi and S. Yamaguchi, "Observation of Method Invocation in Application Runtime in Android for CPU Clock Rate Adjustment," 2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW), 2021, pp. 481-483, doi: 10.1109/CANDARW53999.2021.00090.

[6] K. Kumakura, T. Kamiyama, M. Oguchi, S. Yamaguchi, "Stack-based Method Invocation and Return Monitoring in ART for CPU Clock Rate Adjustment," 2022 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW), 2022.

[7] K. Kumakura, T. Kamiyama, M. Oguchi, S. Yamaguchi, " Investigation of CPU Resource Consumption in Android " 2022 International Conference on Emerging Technologies for Communications (IEICE ICETC 2022), 2022.

[8] K. Kumakura, T. Kamiyama, M. Oguchi, S. Yamaguchi, " CPU Usage Trends in Android Applications," 2022 IEEE International Conference on Big Data (Big Data), Nakanoshima, Osaka, Japan, 2022.

[9] K. Nagata, S. Yamaguchi and H. Ogawa, "A Power Saving Method with Consideration of Performance in Android Terminals," 2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing (IEEE ATC 2012), Fukuoka, 2012, pp. 578-585. doi: 10.1109/UIC-ATC.2012.133

[10] K. Nagata and S. Yamaguchi, "An Android application launch analyzing system," 2012 8th International Conference on Computing Technology and Information Management (NCM and ICNIT), 2012, pp. 76-81.

[11] K. Nishinaka, A. Sonoyama, T. Kamiyama, A. Fukuda, M. Oguchi and S. Yamaguchi, "Monitoring System for Optimization based on Analyzing Android Application Launching Behavior," 2020 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-Taiwan), 2020, pp. 1-2, doi: 10.1109/ICCE-Taiwan49838.2020.9258290.

[12] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi and H. Ye, "Application-Specific Performance-Aware Energy Optimization on Android Mobile Devices," 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 169-180, doi: 10.1109/HPCA.2017.32.

[13] W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, L. N. Pouchet, F Rastello, and P. Sadayappan. 2016. Static and Dynamic Frequency Scaling on Multicore CPUs. ACM Trans. Archit. Code Optim. 13, 4, Article 51 (December 2016), 26 pages. DOI: https://doi.org/10.1145/3011017

[14] Mittal, T., Singhal, L., Sethia, D. (2013). Optimized CPU Frequency Scaling on Android Devices Based on Foreground Running Application. In: Chaki, N., Meghanathan, N., Nagamalai, D. (eds) Computer Networks & Communications (NetCom). Lecture Notes in Electrical Engineering, vol 131. Springer, New York, NY. https://doi.org/10.1007/978-1-4614-6154-8_80

[15] S. Borzsony, D. Kossmann and K. Stocker, "The Skyline operator," Proceedings 17th International Conference on Data Engineering, 2001, pp. 421-430, doi: 10.1109/ICDE.2001.914855.