Vol.2020-MBL-96 No.3
Vol.2020-UBI-67 No.3
Vol.2020-CDS-28 No.3
Vol.2020-ASD-19 No.3
2020/9/29

加速 Android を用いた端末バッテリー消費速度の加速推定

小野里亮祐[1]　神山剛[2]　小口正人[3]　山口実靖[1]

概要： Android 端末の普及に伴い，膨大な数の Android OS 向けアプリケーションが開発，配布されている．アプリケーションのインストールによる端末消費電力の増加量は，アプリケーションの重要な特徴の一つである．アプリケーションによる消費電力の増加量の推定方法として Wakelock の発行量を観察する手法が提案されており，優れた精度で推定をできることが確認されている．しかし，実際にアプリケーションを動作させる動的アプリケーション動作解析による Wakelock 発行量の測定は非常に長い時間がかかることが分かっている．この問題に対して我々は過去の研究において，Android OS 内の Linux カーネルを改変することにより，動的解析に要する時間を短縮する手法を提案している．本稿においては，この手法を適用した加速 Android 環境において短い観察時間で得られた Wakelock 発行数の情報を元に，非加速時の Wakelock 発行数を推定する手法を提案する．具体的には，2 種類の加速倍率で観察を行い，それらの観察値を基に線形近似により非加速時(1 倍時)の発行する手法を提案する．そして，性能評価によりその有効性を示す．

キーワード： 加速テスト，加速 Android，消費電力推定

1. はじめに

Android OS はスマートフォン OS におけるトップシェア[1]であり，アプリケーションストアで配布されているアプリケーションの数も 296 万件と膨大なものとなっている．これに伴って Android OS 向けアプリケーションの動作観察も重要性を増しており，アプリケーション配布サイトの運営者やアプリケーション開発者にとって動作観察は重要なタスクとなっている．アプリケーションの動作観察には様々な観点があるが，特に各アプリケーションがどの程度バッテリの電力を消費するかは，常に持ち歩くスマートフォンやタブレット端末において関心の高い話題であり，アプリケーションの動作解析によるそのアプリケーションの消費電力の調査は重要な課題となっている．

アプリケーションの電力消費量は Android OS の標準アプリケーションの設定(*Setting*)の機能にて調査できるが，複数のアプリケーションで使用される機能による消費電力を各アプリケーションに按分して計上するなど，非常に不正確な数値を報告する実装となっている[2][3]．そこで我々は以前に，アプリケーションによる Wakelock の発行を観察することでアプリケーションが消費した電力をより高い正確さで推定する手法を提案した[4][5]．しかし，この手法にはアプリケーションを実際に動作させての動的観察が必要で，推定に非常に長い時間がかかるという問題点がある．

この課題に対して我々は過去に，アプリケーション動的解析時間の短縮手法を提案した[6]．具体的には，Android OS 内の Linux カーネルを改変し，Android 端末内の時間経過の速度を現実の時間よりも速くすることにより，アプリケーションが認識する時間を加速することでアプリケーションの動作観察時間を短縮するというものである．この手法を適用した環境にてアプリケーションを動作させ動作観察を行ったところ，加速時に Wakelock 発行速度が増加し，消費電力量が増加することや，n 倍の加速度で実行したときの単位実時間あたりの発行数が n 倍にはならない(多くの場合は n 倍未満になる)ことなどが分かった[7]．

本稿では，短い実時間の測定(加速状態における測定)により通常の状態の発行速度(非加速状態における発行速度)を近似的に得る方法として，加速時に観測された Wakelock 発行速度に補正をかけることにより非加速時の Wakelock 発行速度を推測する手法を提案する．そして，その性能評価を行い有効性を示す．具体的には，2 つの加速倍率で観測した Wakelock 発行速度から線形近似で非加速時の Wakelock 速度の推測を行う手法を提案し評価する．

本論文の構成は以下の通りである．2 章にて，カーネルの改変による加速 Android 環境の実現などの Android アプリケーションの動作観察に関する既存の研究を紹介する．3 章にて，本稿の研究にて行った加速時の Wakelock 発行数の調査の結果を示す．4 章にて，前章の結果から加速時における非加速時 Wakelock 発行数の推定手法を提案し，5 章にてその評価結果を示す．6 章にて考察を述べ，7 章にて本稿をまとめる．

2. 関連研究

2.1 Android における時刻管理

Android OS を含む Linux カーネルを用いる OS では，カーネルによって端末内部の時刻が管理され，システム内のプロセスはシステムコール等を使用して時刻の情報をカーネルから取得する．

Linux カーネルではハードウェアから供給されるクロックソースを元に時間情報を管理しており，本稿の実験で使

1 工学院大学.
Kogakuin University
2 長崎大学
Nagasaki University
3 お茶の水女子大学
Ochanomizu University

Vol.2020-MBL-96 No.3
Vol.2020-UBI-67 No.3
Vol.2020-CDS-28 No.3
Vol.2020-ASD-19 No.3
2020/9/29

用した環境では gp_timer もしくは dg_timer がクロックソースとして用いられている．クロックソースから得られた時間情報は変数 cycle_now に格納され，この変数の増分が直前の時刻情報に加算される事で端末内部の時間が経過する．Tickless でないカーネルにおいて時刻情報の更新は tick（Linux カーネルの周期的なタイマ割り込み間隔）ごとに行われ，cycle_now の増分が時刻情報に加算される．Tick 単位は Linux カーネルのパラメータの 1 つであり，コンパイル時に指定することができる．Android OS では多くの場合，tick 単位は 10m 秒（100Hz）とされている．Tickless のカーネルでは更新が tick 毎にはならないが，Tickless でない場合と同様にクロックソースの増分が時時刻情報に加算されることで端末内部時間が経過する．

2.2 動的解析による消費電力の大きいアプリケーションの推定

文献[4][8]にて，アプリケーションの動的解析による消費電力量の大きいアプリケーションの発見手法が提案されている．

同文献では無操作状態のスマートフォンでも起動中のアプリケーションがバッテリを消費することに着目し，無操作状態にて多くのバッテリを消費するアプリケーションを発見する手法について考察している．具体的には，端末がスリープ状態に移行することを妨げ CPU を使用できる状態にしておく Wakelock の発行や，指定時刻に処理を呼び出す Alarm セットの回数の観察などにより無操作状態消費電力を大きく増加させるアプリケーションの発見が可能であることを示している．また当該手法の課題として，実際にアプリケーションを動作させる動的解析には多くの時間を要することが示されている．

2.3 アプリケーション観察時間の短縮

アプリケーションの動的解析の実行時間の短縮手法として，我々は Android OS の Linux カーネル内の時間管理実装を改変し，システム内のアプリケーションが認識する時間の流れを加速することによって観察時間を短くする手法を提案した[6][9]．当該手法では，前述の cycle_now に加算される時刻情報の増分を通常時よりも多くすることによって，端末内の時刻情報の増加速度がクロックソースの増加速度よりも高くなるように修正し，端末内における時間の経過速度を実際の時間よりも高くしている．また，提案手法を実際に Android OS に実装し，実スマートフォン端末にインストールした状態でベンチマークアプリケーションの動作検証を行い，対象のアプリケーションについて正しく加速中の動作観察が実現できることを確認している．

また，本手法をネットワークを用いるクライアント・サーバ型システムのクライアント機ＯＳとサーバ機ＯＳの両方に適用し，クライアント・サーバ型のベンチマークアプリケーションや，実アプリケーションを用いての評価[10][11]を行い，対象のアプリケーションの加速が本手法により正しく行われたことを確認している．

2.4 加速 Android 環境におけるシステム安定性の改善

文献[12][13]にて，加速 Android 環境におけるシステム安定性の改善手法が提案されている．

文献[12]の調査にて，高倍率加速環境下における加速倍率と観察情報の関係性についての評価が行われた．同文献では加速中，無操作状態で端末内時間にて 1 時間 Launcher 画面表示を維持できた回数を計測しシステム安定性を評価している．この調査により，高倍率加速環境下において 120 以上の加速倍率ではシステム安定性が低下すること，Activity Manager やウォッチドッグタイマによるプロセスのキルが発生する事が確認された．加速倍率とシステム安定性の関係を図 1 に示す．

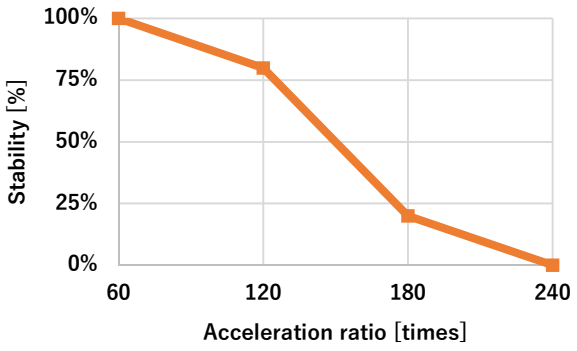

図 1 改善手法適用前の加速倍率とシステム安定性[12]

同文献ではこれらの Activity Manager やウォッチドッグタイマによるプロセスのキルにより Android フレームワークが強制終了され再起動し，加速時のシステム安定性の低下を及ぼしていると考え，Activity Manager によるプロセスキルの無効化，加速倍率 n 倍のときにウォッチドッグタイマによるタイムアウト時間を n 倍に拡大するという対策を行った．この対策の結果，加速倍率 180 まではシステム安定性が向上していることが確認された．結果は図 2 の通りである．

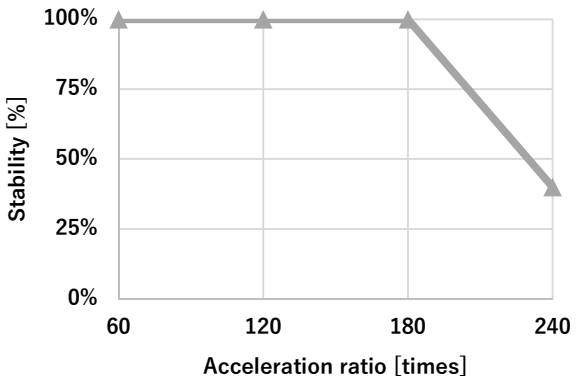

図 2 改善手法適用後の加速倍率とシステム安定性[12]

Vol.2020-MBL-96 No.3
Vol.2020-UBI-67 No.3
Vol.2020-CDS-28 No.3
Vol.2020-ASD-19 No.3
2020/9/29

また，文献[13]にて，前述の手法を適用した後の加速環境下において送信されるシグナルの調査が行われた．同調査では，実時間比 300 倍の加速環境下にて Android 端末を放置し，Android フレームワークが再起動するまでに発行されたすべてのシグナルを Linux カーネルにて記録した．結果として，キルシグナル（signal 9）が発行されていることが確認でき，これによりシステム安定性が低下している可能性があると考えシステムプロセスキルの無効化を行った．同実装では Android OS のシグナル発行部を改変し，発行されたシグナルが 9（キル）かつ対象が Android フレームワークである場合，それを無効化している．

また，同文献ではこのシステムプロセスキルの無効化と文献[12]の手法を実装し，安定性の評価を行った．加速開始時と加速終了時に起動中のプロセスを ps コマンドにて取得し，Android フレームワークプロセスのプロセス ID が変化しているか否かを調査し，端末内時間 1 時間放置後も Launcher プロセスが同じプロセス ID で存在していた割合を安定性として評価を行った．結果は図 3 の通りである．図の Normal は安定性改善手法適用なし，AM+WD は文献[12]の手法適用時，AM+WD+SIG は文献[12]の手法に加えシステムプロセスキルの無効化を行った場合の結果である．図より，文献[12]の手法とシステムプロセスキルの無効化を同時に適用した環境が最も安定性が高いことが分かる．

本論文では，この両手法を同時に適用した加速 Android 環境を使用している．

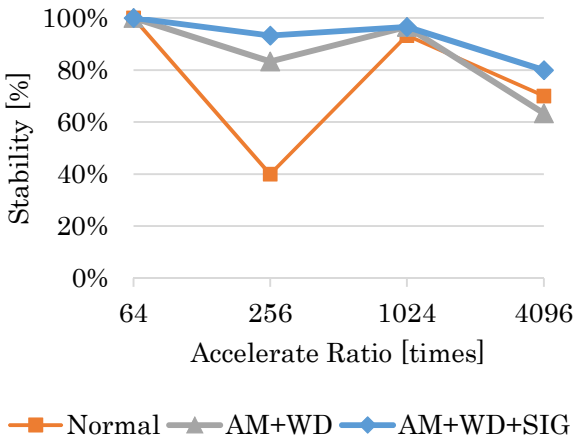

図 3 高倍率加速環境における安定性[13]

また，文献[14]において，システム安定性改善手法を用いた加速 Android 環境における実アプリケーションによる評価が行われている．同文献において，上記のシステム安定性改善手法は実アプリケーションの動作加速においても有効であるということが示されている．

文献[15]にて，加速環境にて TCP 通信を行うアプリケーションの通信安定性の評価と改善手法の提案がされている．同文献にて，加速倍率を増加させると再送などの TCP エラー数が増加すること，再送タイムアウト時間(RTO, Retransmit timeout)を加速倍率に合わせて増加させることにより TCP エラー数の削減が可能であることが示されている．

2.5 加速 Android 環境におけるバッテリ消費量の調査

我々は加速 Android 環境におけるアプリケーションのバッテリ消費量を調査し，加速時は非加速時と比較してバッテリ消費量が 50%～65%程度増加することを示した[7].

ただし，同文献において，Wakelock に着目した調査は行われていない．

3. 加速環境における Wakelock 発行速度の調査

本章にて，加速 Android 環境におけるアプリケーションの Wakelock 発行速度の調査を行う．

実験は，加速 Android 環境をインストールした実端末に評価対象の実アプリケーションを追加インストールし，各アプリケーションを起動し無操作状態で端末を放置，端末のバッテリ残量が 100%の状態から 50%を下回るまでの間の Wakelock 発行を観察することにより行った．端末のバッテリ残量は 5 分おきに確認し，50%を下回っていることが確認された時点で実験終了とした．追加の評価対象アプリケーションセットとしては，2020 年 1 月 26 日付 Google Play Store 無料アプリランキング上位 50 件(*セット 50*), 2020 年 8 月 11 日付 Google Play Store 無料アプリランキング上位 10 件(*セット 10*), 追加インストールアプリケーション無し(*セット 0*)の 3 種類を使用した．*セット 0* においては，AOSP(Android Open Source Project)の Android OS に標準で添付されているアプリケーションのみがインストールされている状況である．Wakelock の発行の確認は，Android OS を改変し Wakelock の発行履歴をテキストファイルでログとして出力するように改変した．計測環境は表 1 の通りである．前述の様に，Wakelock 発行の観察がバッテリ消費を増やすアプリケーションの特定に重要であることが確認されている[4][8]ため，Wakelock に着目して観察を行った．

観察結果は図 1，図 2 の通りである．図 1 はバッテリ残量が 50%以下になるまでの期間に発行された Wakelock 数を表している．図 1 より，どのアプリケーションセットを使用した場合においても，加速倍率の増加に伴って計測期間中の総 Wakelock 発行数も増加していることが分かる．*セット 50* の場合において最大で(加速倍率 5 の例において)約 30%の発行数の増加が確認され，*セット 10* では約 29%，*セット 0* では約 29%であった．

図 2 より，加速倍率の増加に伴って，バッテリ残量が 50%を下回るまでの時間が短くなっていることが分かる．こちらは非加速時と比較した場合，加速倍率 2 倍では約 25%の減少，加速倍率 5 倍では約 58%の減少となっている．また，加速倍率 1 倍から 3 倍までの変化は大きいものの，3 倍か

Vol.2020-MBL-96 No.3
Vol.2020-UBI-67 No.3
Vol.2020-CDS-28 No.3
Vol.2020-ASD-19 No.3
2020/9/29

ら 5 倍までの変化は比較的小さいことが分かる．また，アプリケーションを起動する数が多いほどバッテリ残量が 50%を下回るまでの時間が短いことが分かる．

表 1 計測環境

使用端末	Nexus 7 (2013)
OS	Android 6.0.1 (AOSP) 加速改変済み

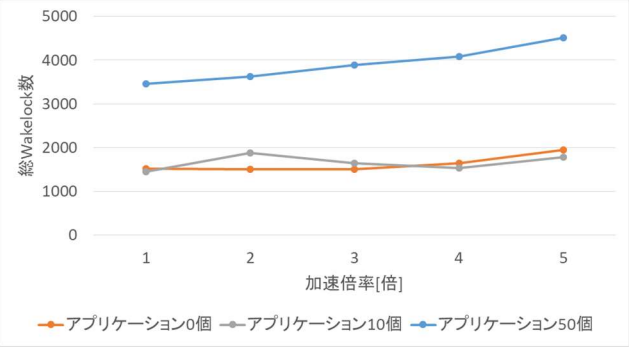

図 4 各加速倍率における総 Wakelock 発行数

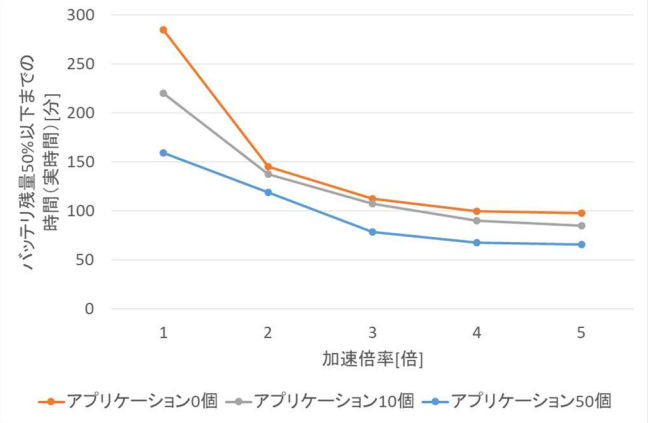

図 5 各加速倍率におけるバッテリ残量 50%以下までの時間

4. 非加速時 Wakelock 発行速度の推定手法

本章にて，非加速時 Wakelock 発行速度を加速 Android 環境における観察結果より推定する手法を 2 つ(アプリケーション分類なし手法とアプリケーション分類あり手法)提案する．

4.1 アプリケーション分類なし手法

アプリケーション分類なし手法は，2 つの加速倍率（例えば，2 倍と 5 倍）で取得した Wakelock 発行数のデータから，線形近似により非加速時の推定 Wakelock 数を求める．推定値は以下の式で求める．

加速倍率 a と加速倍率 b でそれぞれ m 回, n 回の Wakelock が発行された時，加速倍率 x での Wakelock 発行数 y は

$$y = \frac{n-m}{b-a} \times (x-a) + m \quad \cdots (1)$$

となる。(1)式より，加速倍率 1 倍（非加速時）の y は

$$y = \frac{n-m}{b-a} \times (1-a) + m$$

となる．

4.2 アプリケーション分類あり手法

アプリケーション分類あり手法は，前項の分類なし手法と同様に 2 つの加速倍率で取得した Wakelock 数より非加速時の Wakelock 数を推定する．この手法ではアプリケーションを 2 つのグループに分類する．加速した際に発行された Wakelock 数の 2 つの加速倍率間における増加比率が特定の閾値を超えたアプリケーションをグループ A，そうでないアプリケーションをグループ B とする．すなわち，加速倍率 a と加速倍率 b でそれぞれ m 回，n 回の発行があったとき(ただし $b>a$ とする)，n/m>閾値を満たす場合はグループ A とし，そうでない場合はグループ B とする．グループ A の Wakelock 数は分類なし手法と同様に線形近似で推定し，グループ B の Wakelock 数は 2 つのデータからの平均値で算出する．

上記の 2 個の提案手法は，主として以下の様な考え方に基づいている．まず，無操作状態の端末におけるアプリケーションの動作やそれに伴う Wakelock 発行の頻度(実時間あたりの回数)は，端末内部の時計の加速により同等以上になるとの仮説に基づいている．そして，動作や Wakelock 発行は端末内時間に基づいている場合とそうでない場合があると考えられる．前者の例としては，端末内部で時間を計り，(15 分に 1 回新着メールを確認する様な)定期的にサーバに接続する処理を行う様なものが考えられる．このような動作の発生速度(単位時間あたりの実行回数)は，端末内時間を n 倍に加速することにより n 倍になると予想される．後者の例としては，端末外部の時計で時間を計り，定期的にクライアント端末に通知を行いクライアント端末で処理が行われる様なものが考えられる．この場合，端末内時間の流れる速さが n 倍になっても発行速度は 1 倍のままであると予想される．そして，アプリケーションによっては前者と後者の両方の側面を持っており，例えば全発行の p%は加速で n 倍になり，100-p%は加速しても 1 倍のままであるなどが考えられる．この様な例においては，a 倍加速時と b 倍加速時の速度から線形近似することにより x 倍加速時の速度を推定できると考えられる．一方で，観測データが上記の様な仮説に基づかず，加速倍率 a と b のときの発行回数 m と n が $a<b$ かつ $m>n$ の様な関係になった場合は，線形補間を行うと加速時よりも高い速度での非加速時における発行を推定する．アプリケーション分類あり手法ではこれを不適切と予想し，線形補間を用いず m と n の間から推定値を選定している．

5. 非加速時 Wakelock 速度推定手法の評価

本章にて，前述の非加速時 Wakelock 発行速度の推定手法の評価を行う．

Vol.2020-MBL-96 No.3
Vol.2020-UBI-67 No.3
Vol.2020-CDS-28 No.3
Vol.2020-ASD-19 No.3
2020/9/29

評価は第 3 章で計測した加速 Android 環境下での Wakelock 発行速度から，第 4 章の 2 種類の手法を用いて非加速時の Wakelock 発行速度の推測を行い，正解値との比較により行う．推定結果と正解値を図 6～図 10 に示す．横軸は推定のためのサンプルとして使用した加速倍率の組み合わせ，縦軸が推定 Wakelock 発行速度（端末内時間 1 時間あたりの推定 Wakelock 発行回数）である．図 6 から 8 の分類あり手法の結果は閾値 1.0 を用いたものであり，図 9,10 における結果は閾値 1.0 から 2.0 まで 0.1 ずつ変化させ，最も低い誤差で推定できた閾値による推定結果を抽出したものである．ただし，*セット 0* における最も低い誤差の閾値は全て 1.0 となり，図 6 と同様の結果になったため掲載を省略する．

まず，図 6 から 8 の閾値に 1.0 を用いた場合について述べる．図 6,7 よりアプリケーション 0 個の *セット 0* の環境下では分類あり手法が最小約 2.3%，最大約 48%，分類なし手法が最小約 0.2%，最大約 14%となっており，アプリケーション 10 個の *セット 10* の環境下では正解値との誤差が分類あり手法が最小約 11%，最大約 43%，分類なし手法が最小約 2.1%，最大約 30%となっていることが分かり，分類なし手法の方が精度が高い結果となっていることが分かる．ただし，分類あり手法と分類なし手法では低い誤差で推定できた加速倍率の組み合わせが異なり，分類あり手法では[2,3][2,4][2,5]倍の組み合わせ，分類なし手法では[3,4][3,5]倍の組み合わせの誤差が低い傾向にある事が分かる．特に，[2,3][2,4][2,5]倍の組み合わせにおいては，分類なし手法では推定値が正解値よりも大きくなっていることが分かる．従って，必ずしも分類なし手法の方が優れているとは言えず，分類あり手法の方が優れている場合と分類なし手法の方が優れている場合があるということが分かった．一方で，図 8 より，アプリケーション 50 個の *セット 50* の環境下では分類あり手法が最小約 23%，最大約 44%，分類なし手法が最小約 6.9%，最大約 35%となり，全ての加速倍率の組み合わせにおいて分類なし手法の方が精度が高い結果となった．

次に，図 9 から 10 の最も良い推定精度となった閾値の結果について述べる．図 9 より，*セット 10* の環境下では分類あり手法の正解値との最小誤差が閾値 1.0 の時より僅かに改善し，約 10%となったことが分かる．また，図 10 より，*セット 50* の環境下では精度が低かった組み合わせ[2,3]倍での精度が 44%から 28%まで改善され，分類あり手法の最大誤差が約 33%まで減少した事が分かる．この加速倍率組み合わせでは，分類あり手法が分類なし手法よりも高い精度という結果になっている．

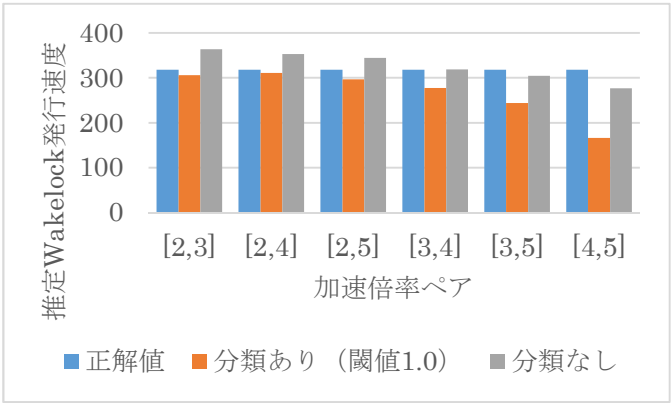

図 6 アプリケーション 0 個での評価

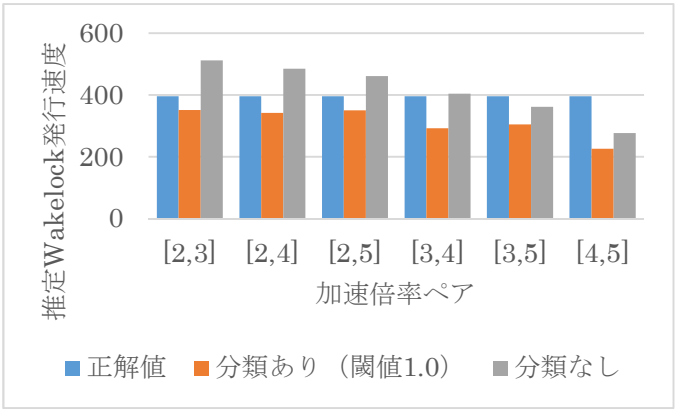

図 7 アプリケーション 10 個での評価

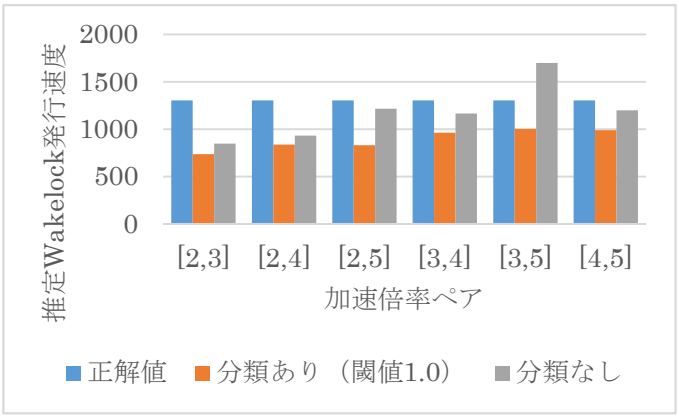

図 8 アプリケーション 50 個での評価

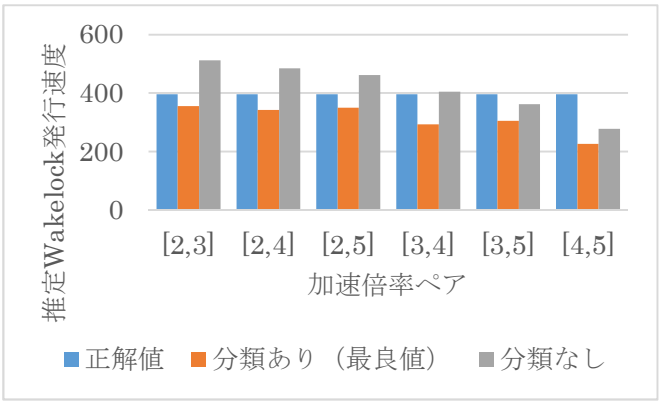

図 9 アプリケーション 10 個での評価（最良値）

Vol.2020-MBL-96 No.3
Vol.2020-UBI-67 No.3
Vol.2020-CDS-28 No.3
Vol.2020-ASD-19 No.3
2020/9/29

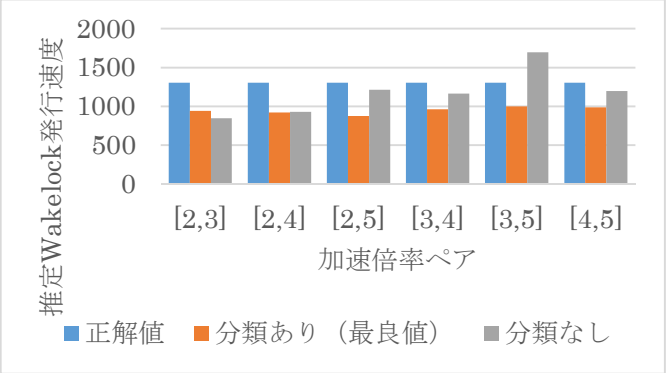

図 10 アプリケーション 50 個での評価（最良値）

6. 考察

本章にて，両提案手法における推定 Wakelock 発行速度の差の原因と，実環境における両提案手法の適用の適切さについての考察を行う．5 章にて述べたように，*セット 10* や *セット 0* の環境下では一部の加速倍率組み合わせにおいて，分類なし手法では推定値が正解値よりも大きくなった．また，すべての場合において，分類あり手法は分類なし手法よりも低い推定値となった．このような結果になった理由としては，4.2 節で述べたように加速倍率 a と b のときの Wakelock 発行回数 m と n が $a<b$ かつ $m>n$ の様な関係になるアプリケーションが存在し，そういったアプリケーションについての非加速時 Wakelock 発行速度の推定値は線形補完により正解値よりも大きな値となり，加速時よりも高い速度での非加速時における発行を推定してしまっているためだと考えられる．これを修正するには，分類あり手法が有効であると期待できる．ただし，現状では分類あり手法は閾値を適切に選択しないと良い精度が得られておらず，このようなアプリケーションに対してはさらなる考察が必要であると考えられる．

また，実際の使用環境を想定した場合，50 個という大量のアプリケーションを同時に起動する *セット 50* の状況は多くないと考えられ，*セット 10* や *セット 0* の方が実際の環境に近いと考えられる．*セット 10* や *セット 0* の結果においては分類あり手法が優れている場合と分類なし手法が優れている場合があるという結果になったが，上記の理由から分類なし手法では過剰に Wakelock 発行速度を大きく推定してしまう可能性があるため，適切な閾値設定を行うことができれば分類あり手法の方が誤差の小さい推定を行えると考えられる．よって，本手法を実ユーザ端末におけるアプリケーション Wakelock 発行速度の推定に用いる場合，適切な閾値の設定方法の検討が重要であると考えられる．

7. おわりに

本稿では，加速 Android 環境における加速時の Wakelock 発行速度増加に着目し，加速 Android 環境下での Wakelock 発行速度の観察データから非加速時の Wakelock 発行速度を推測する手法を提案し，その評価を行った．具体的には，加速時の Wakelock 発行速度の増加比率から各プロセスをグループ分けし，線形近似と相加平均から非加速時の Wakelock 発行速度を求める手法を適用することで，最小誤差 2.3%～23%の精度で非加速時の Wakelock 発行速度を推定することが可能であることを示した．

今後は，アプリケーション分類あり手法における閾値の設定方法の検討などを行う予定である．

謝辞

本研究は JSPS 科研費 17K00109, 18K11277 の助成を受けたものである．

本研究は，JST，CREST JPMJCR1503 の支援を受けたものである．

参考文献

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

[2] Shun Kurihara, Shoki Fukuda, Takeshi Kamiyama, Akira Fukuda, Masato Oguchi, Saneyasu Yamaguchi, “Estimation of Power Consumption of Each Application Considering Software Dependency in Android,” Journal of Information Processing, 2019, Volume 27, Pages 221-232, Released February 15, 2019, Online ISSN 1882-6652. doi: 10.2197/ipsjjip.27.221

[3] Shun Kurihara, Shoki Fukuda, Shintaro Hamanaka, Masato Oguchi, Saneyasu Yamaguchi, “Application power consumption estimation considering software dependency in Android,” In Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication (IMCOM '17), Association for Computing Machinery, New York, NY, USA, Article 86, pp. 1–6, 2017. doi: 10.1145/3022227.3022312

[4] Shun Kurihara, Shoki Fukuda, Saneyasu Yamaguchi, Ayano Koyanagi, Masato Oguchi, Ayumu Kubota, Akihiro Nakarai, "A study on identifying battery-draining Android applications in screen-off state," 2015 IEEE 4th Global Conference on Consumer Electronics (GCCE), 2015, pp. 603-604, doi: 10.1109/GCCE.2015.7398682.

[5] S. Kurihara, S. Fukuda, S. Hamanaka, M. Oguchi and S. Yamaguchi, "Identifying battery-draining applications by monitoring behavior in screen-off state in Android," 2016 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW), Nantou, 2016, pp. 1-2, doi: 10.1109/ICCE-TW.2016.7520971.

[6] S. Fukuda, S. Kurihara, S. Hamanaka, M. Oguchi and S. Yamaguchi, "Accelerated application monitoring environment of Android," 2016 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW), Nantou, 2016, pp. 1-2, doi: 10.1109/ICCE-TW.2016.7520972.

[7] Ryosuke Onozato, Takeshi Kamiyama, Akira Fukuda, Masato Oguchi, Saneyasu Yamaguchi, "Observation of Power Consumption of Each Application in an Accelerated Android OS," 2020 IEEE ICCE-Taiwan, 2020.

[8] 栗原 駿，福田翔貴，小柳文乃，小口正人，山口実靖 “Alarm の観察による無操作状態携帯端末の消費電力の増加の原因となるアプリケーションの推定”，信学技報, vol. 115, no. 230, DE2015-23, pp. 17-22, 2015 年 9 月．

[9] 福田翔貴，栗原駿，濱中真太郎，小口正人，山口実靖, "Android アプリケーション観察の加速環境の構築", 情報処

Vol.2020-MBL-96 No.3
Vol.2020-UBI-67 No.3
Vol.2020-CDS-28 No.3
Vol.2020-ASD-19 No.3
2020/9/29

理学会 研究報告コンシューマ・デバイス＆システム（CDS），Vol. 2016-CDS-15，No. 27, pp. 1-8, 2016.

[10] S. Fukuda, S. Kurihara, S. Hamanaka, S. Yamaguchi and M. Oguchi, "An accelerated application monitoring environment with accelerated servers," 2016 IEEE 5th Global Conference on Consumer Electronics, Kyoto, 2016, pp. 1-2, doi: 10.1109/GCCE.2016.7800527.

[11] 福田翔貴，栗原駿，濱中真太郎，小口正人，山口実靖, "Android アプリケーション観察を目的としたクライアント・サーバ加速環境”，信学技報, vol. 116, no. 214, DE2016-16, pp. 25-30, 2016 年 9 月.

[12] 福田翔貴，栗原駿，濱中真太郎，小口正人，山口実靖，“時間加速 Android 環境におけるシステム安定性に関する一考察”，16 回情報科学技術フォーラム(FIT2017)，B-019

[13] 福田翔貴，栗原駿，神山剛，福田晃，小口正人，山口実靖，“加速 Android 環境におけるシステム安定性の改善”，情報処理学会 研究報告コンシューマ・デバイス＆システム（CDS）, Vol. 2018-CDS-21, No. 31, pp. 1-6, CDS21-22

[14] 小野里亮祐，福田翔貴，神山剛，福田晃，小口正人，山口実靖, "時間加速 Android 環境のシステム安定性のアプリケーションによる評価”，情報処理学会 研究報告コンシューマ・デバイス＆システム（CDS），巻 2018-CDS-22，号 11，pp. 1 － 8，2018 年 5 月

[15] Ryosuke Onozato, Akira Fukuda, Takeshi Kamiyama, Masato Oguchi and Saneyasu Yamaguchi: “Reducing TCP Errors in Accelerated Application Test in Android OS”, IEEE INTERNATIONAL CONFERENCE ON CONSUMER ELECTRONICS – TAIWAN (IEEE 2019 ICCE-TW),2019