情報処理学会論文誌 Vol.57 No.2 394–405 (Feb. 2016)

1フレーム分の描画処理終了推定によるGPU状態制御

野呂 正明[1,a)] 村上 岳生[1] 上和田 徹[1] 石原 輝雄[1]

受付日 2015年4月23日, 採録日 2015年11月5日

概要：多くのスマートフォンのユーザは端末の電池持続時間を重視している一方で，CPU や液晶ディスプレイと比較して，GPU の消費電力削減の研究開発はそれほど盛んではない．Android 端末の GPU は，1回の画面更新の期間に，処理がある期間と処理がないアイドル期間が存在するが，アイドル期間中は短時間で処理再開可能であるが電力の大きな状態に設定されている．一方，Android では，1 回の画面更新に必要な描画の終了から，次の描画期間の開始までまとまった処理がないアイドル期間が現れる確率が高い．そのため，1 回の画面更新の描画処理がすべて終了したことを判定できれば，GPU を低電力状態に設定することが可能となる．本研究の提案手法では，実行中のアプリケーションが更新する仮想画面の数の履歴から，次の描画期間に発生する仮想画面の更新数を推定し，その数の仮想画面の更新が終了した時点で GPU を低電力状態に設定する．Nexus5 用にプロトタイプを開発し，実際に動作させて性能評価を行ったところ，アプリケーションの種類によるが，端末の消費電力を 1.8%から 4.7%削減できた．

キーワード：モバイル，ユビキタスコンピューティング，低消費電力化技術

GPU State Control Method with Estimating Termination of Drawing

Masaaki Noro[1,a)] Takeo Murakami[1] Toru Kamiwada[1] Teruo Ishihara[1]

Received: April 23, 2015, Accepted: November 5, 2015

Abstract: About 40% of users select mobile phone by battery life. GPU is one of power cosuming deivce of mobile phone. The GPU in Android phone changes their state (“active” and “idle”) during drawing screen. There are three kinds of idle state (clock gating, partially power gating and power off). But, GPU of mobile phone uses only most power consuming state (clock gating) while drawing idle time.Many Android applications uses GPU very shorter time than the one frame time. GPU power consumption can be reduced by changing GPU state to low power state after finish drawing.We propose the method that estimates end of one frame drawing using number of drawing in one frame time. We evaluate our method by prototype that runs on Nexus5. Proposed method can reduce 1.8% to 4.7% of terminal power consumption.

Keywords: Mobile computing, Ubiquitous computing, Low-power consumption techniques

1. 背景

Android [1][*1]を搭載した端末をはじめとして，スマートフォンの消費電力はフィーチャーフォンと比較して大きい．さらに，無線の電力増加 [2]，画面サイズの大きな端末の流行といった消費電力を増加させる要因もある．そのため，スマートフォンにおける電池の持続時間に対するユーザニーズは高く，40%以上が購入時に電池持ちを重視しており [3]，スマートフォンの消費電力削減は重要な課題である．

過去，ノート PC ではディスプレイ，ハードディスク，CPU が主に電力を消費しており，さまざまな研究が行われてきた．たとえば CPU では，処理がない場合に回路の一部（もしくはすべて）に対するクロックや電源の供給を停止する（CPU の状態を変化させる）ことで CPU が処理を行っていない場合の消費電力を削減することができ

[1] 富士通研究所
FUJITSU LABORATORIES LTD., Kawasaki, Kanagawa 211–8588, Japan
[a)] noro@jp.fujitsu.com

[*1] Android は Google Inc. の登録商標である．

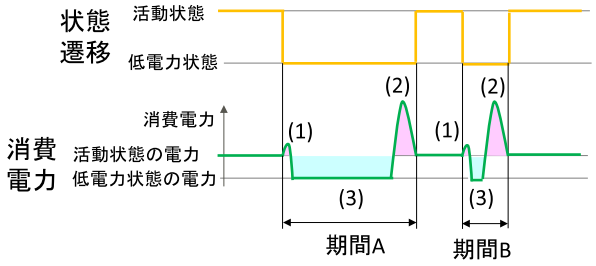


図 1 状態制御と消費電力

Fig. 1 State transition and Power consumption.

る．この際，プログラムの実行履歴に基づいて，処理が終了した時点で CPU をどの状態に設定するか判定する手法（Linux[*2]の cpuidle [4]）が利用されている．また，ハードディスクも，アクセスがない場合にディスクの回転を停止する手法 [5] がある．

スマートフォンではハードディスクが搭載されておらず，CPU は cpuidle が活用されており，液晶ディスプレイの消費電力削減も進められている．しかし，GPU の消費電力削減はこれまであまり行われてこなかった．現在のところ，Nexus5[*3] [6] で youtube 視聴時と chrome ブラウザで web ページを閲覧した際の消費電力の平均値を測定したところ，1 W と 2.2 W であったのに対して，Nexus5 の GPU で，処理がない期間にクロック供給を止めた場合と，回路の一部への電源供給を止めた場合の端末の電力差は，GPU の動作周波数によって異なるが，周波数最小で 64 mW，最大で 402 mW（詳細は 4.5 節を参照）であり，端末の電力に対する比率は youtube の消費電力の 6.4%と 40.1%，chrome ブラウザの動作電力の 2.9%と 18%となり，処理がない期間に GPU の状態を制御することが端末の消費電力削減につながる．

ただし，GPU や CPU の状態（電源やクロックの供給状況）遷移により消費電力が増加する場合がある．**図 1** は処理がない期間に CPU や GPU を低電力状態に遷移する場合における装置全体の消費電力を示している．CPU や GPU を活動状態から低電力状態に遷移する場合（図の (1)）はクロックや電源の制御のために消費電力が増加する．また，処理が発生して低電力状態から活動状態に遷移する際（図の (2)）はクロックや電源の制御だけでなく，回路に電荷を蓄積する必要があるため，非常に多くの電力を消費する．そのため，GPU や CPU の消費電力を削減するために，処理がない場合に状態を変更する場合，(1) の電力 + (2) の電力 = (3) の電力 となる期間の長さより長時間，処理がない状況が続く必要がある．図 1 の例では，期間 A は消費電力を削減できているが，期間 B は消費電力が増加する．

[*2] Linux は，Linus Torvalds 氏の日本およびその他の国における登録商標または商標である．
[*3] Nexus は Google Inc. の登録商標である．

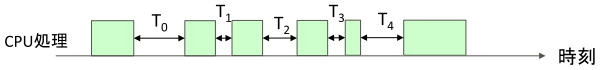


図 2 CPU 処理の例

Fig. 2 CPU processing example.

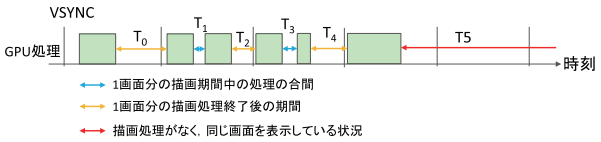


図 3 GPU 処理の例

Fig. 3 GPU processing example.

CPU では状態変更による消費電力削減を実現するため，状態遷移ロス（図 1 における (1) や (2) の期間で電力が増加する量）や消費電力が異なる複数の状態が提供されており，cpuidle は最大 10 個の状態を取り扱うことができる．**図 2** は CPU 処理の事例である．cpuidle はタスクスケジューラと連動しているため，次のタスク割付けまでの残り時間（図 2 の T_0 から T_4 の長さ）は既知である．すると，CPU 処理がない期間（T_0 から T_4）に設定すべき CPU の状態は設定カーネル内部の cpuidle モジュールで判断することができる．そのため，開発者は Linux を新たな CPU に移植する際，CPU 処理がないアイドル期間の長さと CPU 状態の組合せを定義することで，cpuidle は CPU の状態を制御することにより消費電力を削減する．

一方，GPU はある描画処理が終了し，GPU が実行する処理がない状態（以後 “アイドル状態” と呼ぶ）となった時点で，次の GPU 処理が発生するまでの時間（**図 3** における T_0 から T_5 の長さ）を判断することが困難であるため，cpuidle の手法は利用できない．状態遷移による消費電力の削減を実現するためには，アイドル状態の期間（以後 “アイドル期間” と呼ぶ）の長さを判定する手法が必要である．現在のところ，アイドル期間の長さを判定する一般的な手法が存在しないため，GPU の状態制御による電力削減は多くのプラットフォームにおいてあまり活用されていない．そのため，Linux では GPU の状態制御による省電力化の共通のプラットフォームも存在せず，個々のドライバ開発者に委ねられている．たとえば，メインライン [7] や主要ベンダ [8], [9] の実装では，ON，OFF の 2 種類，Google による Nexus5 用の実装 [10] では ON，OFF に加えて，クロックの供給停止状態と，回路の一部への電力供給停止の 4 種類が利用可能である．cpuidle と比較して利用可能な状態数は少ないものの，一定以上の長さのアイドル期間を発見して GPU を低消費電力の状態に設定することができれば，端末の消費電力削減が期待できる．たとえば，Nexus5 の実装では図 3 の 3 種類のアイドル期間のうち，「1 画面分の描画期間中の処理の合間」と「1 画面分の描画処理終了後の期間」は同じ状態と見なされ GPU へのクロック供給が停止され，「描画処理がなく，同じ画面を表示している

状況」は GPU が OFF となる（詳細は 2.1 節参照）．ただし，「1 画面分の描画期間中の処理の合間」の長さと「1 画面分の描画処理終了後の期間」の長さを比較すると「1 画面分の描画処理終了後の期間」が長いため（詳細は 2.4 節参照），「1 画面分の描画処理終了後の期間」の GPU 状態をより消費電力が少ない状態に設定することができれば，端末の消費電力削減に貢献できる．以上のような理由から，本研究ではスマートフォンを対象に，「1 画面分の描画期間中の処理の合間」と「1 画面分の描画処理終了後の期間」を識別し，GPU の状態を制御することで，消費電力を削減する手法を提案する．

2. 従来手法

ここでは，現在の Android の描画の仕組みと，GPU の状態を制御して端末の消費電力を削減するための手法について述べる．

2.1 Android の描画方法と GPU 制御

既存の Android 端末のうち，GPU の状態制御を行っている機種（Nexus5）は，2 種類の状態制御（**図 4**）を行う．1 つ目は，画面に静止画が出た状態で GPU アイドル期間が長時間（少なくとも数十 ms 以上）続く場合に GPU の電源を落とす方法であり，2 つ目は GPU 処理間の短いアイドル期間に状態を変化させ，電力を削減する方法である．なお，図 4 において，GPU 状態の，「活動」状態は GPU が処理を行っている状態，「待機」状態は処理はなく，回路に電源が供給されているが，クロックの供給は止まっている状態，「休止」状態は回路の一部に対する電源供給が止まっている状態，「電源断」状態は全回路の電源供給を止めた状態である．ただし，待機状態や休止状態において，どの範囲まで供給が行われているかについては，チップの種類やカーネルのバージョンによって異なる．

また，図 4 における VSYNC は画面の垂直同期信号が発生するタイミングである．ディスプレイは，一定周期（1/60 秒もしくは，1/30 秒）で画面の表示を切り替えるが，この表示の切替えのタイミングを画像を作る装置と実際に表示するディスプレイで合わせなければならない．そのためのタイミング通知に用いる信号が垂直同期信号である．

同じ画面を表示したままユーザが端末画面を見続けるような場面では，GPU は処理がない状態で電力を消費し続けることとなる．1 つ目の方法では，このような場合を検出するため，GPU の処理終了から一定時間以上新しい GPU 処理（描画）が発生しなかった場合にタイマでカーネルが GPU を「OFF（電源断）」状態に設定することで電力を削減する．

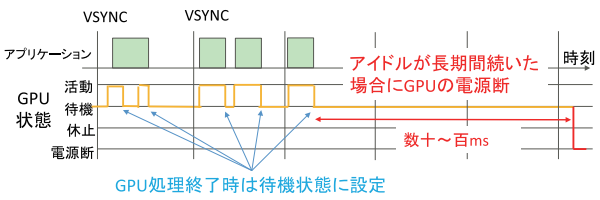


図 4 描画と状態制御の例
Fig. 4 Drawing and GPU state transition on Android.

次に，2 つ目の方法について述べる．Android の画面は複数の仮想画面から成り立っており（**図 5**），各アプリケーションは 1 枚以上の仮想画面をシステムから獲得し，その仮想画面上に自分の絵を描いたうえで，仮想画面描画終了を OpenGL の特定の機能を呼び出すことでシステムに通知する．すると，次回の画面合成時に描画済みの仮想画面が重ねあわせ処理され，画面に反映される（**図 6**）．

描画処理は，3 段階のパイプライン（**図 7**）で構成され，パイプラインの各段階は 2 つの VSYNC の間（1 回の描画期間）に実行する．アプリケーションは新しい描画期間開始の通知を受信すると描画を実行し，その終了をシステムに通知する（パイプラインの第 1 段）．システムはアプリケーションの描画終了通知を受けた後，次の描画期間の開始を検出（VSYNC の受信）すると，第 1 段の描画結果を画面出力する画像に合成し，アプリケーションに新しい描画期間開始を通知する（第 2 段）．3 番目の描画期間の開始（VSYNC の発生）をディスプレイコントローラが検出し，

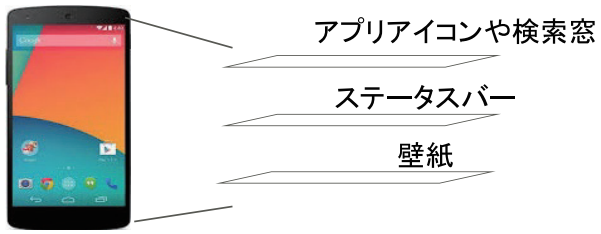


図 5 Android の画面構成
Fig. 5 Virtual screen of Android.

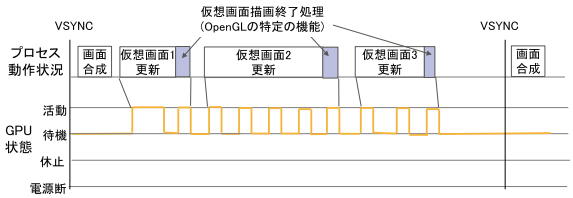


図 6 Android の画面描画
Fig. 6 Screen drawing on Android.

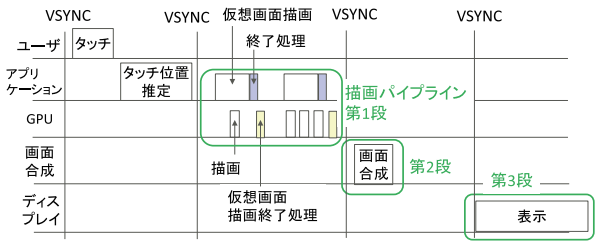


図 7 描画のパイプライン
Fig. 7 3 stage pipeline of Android graphic system.

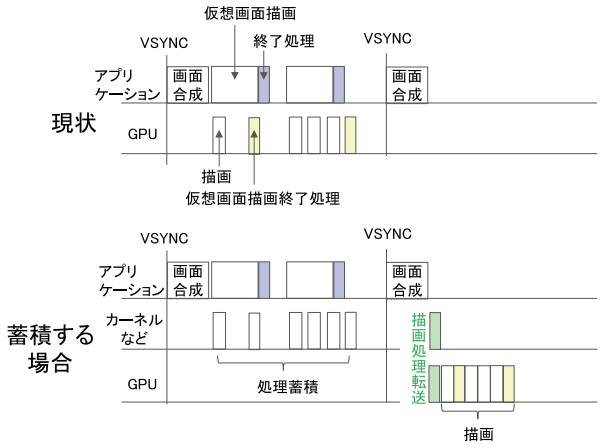


図 8 GPU 処理を蓄積して一括実行する例
Fig. 8 An example of one time GPU processing.

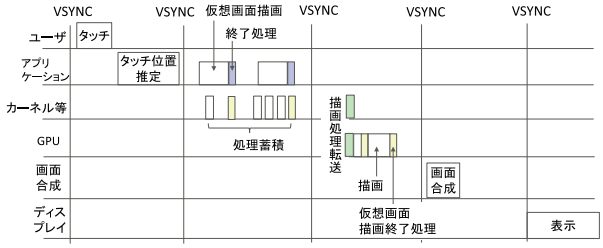


図 9 描画処理を蓄積する場合のパイプライン
Fig. 9 Pipeline of one time GPU processing.

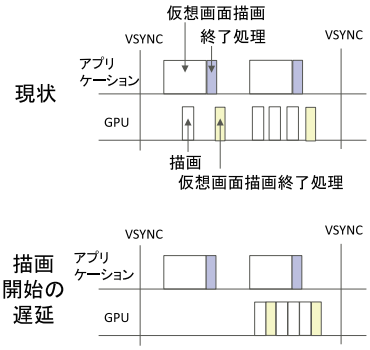


図 10 描画処理の開始を遅らせる方法
Fig. 10 Delayed starting method.

合成された画像を画面に表示する（第 3 段）．なお，ユーザの画面操作で描画が行われる場合は，ユーザの画面操作の次の描画期間にアプリケーションの描画が行われる．パイプラインの第 1 段の描画では，複数のアプリケーションが任意のタイミングで GPU を利用することが可能であり，描画を行うアプリケーションは GPU の予約等を事前に行う必要はない．すると，GPU の処理が終了した時点で，この期間に描画を行うアプリケーションか残っているか否かをカーネルから判断することができない．

そのため，GPU 処理が終わりアイドル期間に入る際に，後続処理が短時間で発生した場合でも消費電力が増加しないよう，状態遷移ロスは少ないが，比較的消費電力の大きな状態（図 6 における「待機」状態）に設定している．

2.2 描画処理の蓄積

現在の Android では，アプリケーションが発行した GPU への描画処理依頼は，カーネルを通じてただちに GPU に転送され，GPU は処理を実行するため，**図 8** における「現状」の図のようにアイドル期間が細切れに発生する．それに対して，図 8 の「蓄積する場合」のように，GPU に依頼する処理をカーネル等で一度蓄積し，次の描画期間に GPU で一括処理する方法が考えられる．この場合，現在より GPU のアイドル期間は長くなり，待機状態より消費電力が少ない休止状態や電源断状態に設定することができる可能性がある．

しかし，この方法では**図 9** のように描画パイプライン段数が 3 から 4 に増加し，ユーザによる画面タッチを契機に処理が進むアプリケーションにおけるタッチ位置推定（図 9 のアプリケーションの動作）で問題が発生する．タッチ位置推定は指の現在位置ではなく，過去の指の動きから画面に絵が出るタイミングで指が存在する場所を推定する．このため，パイプラインの段数が増えた場合，アプリケーション自ら指の動きを解析するアプリケーション（ゲーム等）は GPU 処理の蓄積を行う機種と，その他の機種でプログラムを変更する必要生じる．マーケットの多くのアプリケーションが一部の機種への対応のために，このような変更を加えることを期待することは難しいため，GPU 処理を蓄積する方式は本研究では採用しない．

2.3 描画開始時間の遅延

図 10 の「描画開始の遅延」のように，GPU 処理の開始を故意に遅らせ，すべての GPU 描画処理が 1 度で終わるように調整する方法が考えられる．この方法では，1 フレーム分の描画処理の量や，処理が発生する時間を推定したうえで GPU の処理開始をどの程度遅延すればよいか判断する．この方法には，GPGPU 分野で研究されている処理のスケジューリング [11], [12] や省電力 [13], [14], [15], [16], [17], [18] の研究成果が適用できる．

しかし，GPU 処理の実行開始を遅延させた場合，描画が 1 回の描画期間で終了せず，描画のフレームレートが低下する危険が大きい．描画フレームレートはベンチマークやゲームの性能に大きく影響するため，フレームレート低下の危険が大きい方法を採用することは難しい．

2.4 1 フレーム分の描画終了を契機とした GPU 状態制御

2.1 節で，現在の Android 端末における GPU 状態制御について説明したが，主要なアプリケーションの動作のログに基づいて詳細に説明する．本研究では，動作ログ取得のため，Nexus5 用 Android4.4.2 の Linux カーネルにおける GPU ドライバを拡張し，GPU に処理が投入された時

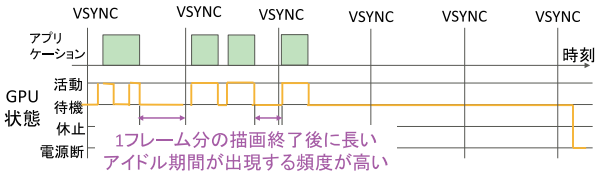


図 11 描画終了後のアイドル期間
Fig. 11 GPU idle after drawing.

刻，投入された処理が終了した時刻，GPU の状態変化をログに出力した．また，アプリケーションが保持する仮想画面に対する描画処理の終了を示す OpenGL の API 呼び出しと VSYNC の信号を検出するため，Android フレームワークにもログ出力機能を追加した．

なお，この実験におけるアプリケーションは携帯電話事業者の実使用時間のシミュレーション [19] に用いられるアプリケーションのうち，消費電力の占める比率（アプリケーションの消費電力とシミュレーションにおける割当ての両方）が大きい「Android ブラウザ，Google Chrome，youtube」および，市場調査や実使用時間のシミュレーションには現れないが，アプリの立ち上げや基本的な端末操作の際にユーザが必ず利用することと，Android SDK を用いたアプリの典型的な実装となっていることから，ホーム画面アプリケーション（以後，ホームアプリと呼ぶ）を採用した．ここで，ブラウザ 2 種類を利用した理由は，Android version 5 以降の Android では，OS に HTML レンダリングエンジンが搭載されず，ブラウザと HTML レンダリングエンジンがアプリストア経由で配布されるものの，古いバージョンの OS や一部のベンダの端末では，従来の Android ブラウザと OS プレインストールの HTML レンダリングエンジンを利用するためである．

なお，各アプリの操作の詳細は次のとおり．youtube は横長全画面表示で動画を約 3 分間再生した．また 2 つのブラウザは，1 秒に 1 回程度画面を上下にスクロールさせたうえで，別のページを開き，再度スクロールするという手順を 1 分間繰り返した．ホームアプリでは，1 秒に 1 回程度手動で画面をスクロール（もしくはスワイプ）する動作を 1 分間繰り返した．

収集したログを分析したところ，GPU の 1 画面の描画終了後のアイドル期間の長さは 7 ms 以上となることが多く（**図 12** 参照），**図 11** のように，Android では 1 フレーム分の描画処理がすべて終了した後に，GPU が比較的長いアイドル期間となる確率が高い．

現在の Android 端末では，1 フレーム分の描画がすべて終了したか否かを判定することが困難であるため，アイドル期間が短い場合でも，GPU の状態遷移による消費電力の増加が発生しない「待機」に設定されている．一方，1 つの描画期間に実行される 1 フレーム分の描画終了を識別し，次の描画期間の開始までの時間が判定できれば，「休止」状態や「電源断」状態に設定できる可能性がある．

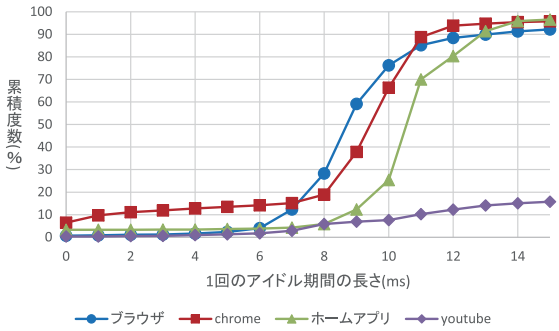


図 12 端末のアイドル期間の長さ
Fig. 12 Length of GPU idle.

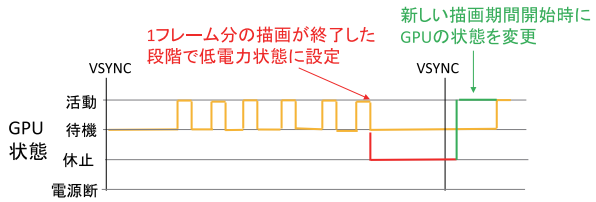


図 13 描画終了後に状態を変更する方法の例
Fig. 13 An example of GPU state transition.

文献 [20] は，1 つのアプリケーションによる描画の終了時に発生するイベントをとらえて，それを 1 フレーム分の描画の終了と見なし，GPU を低電力状態に設定する方法（**図 13**）を提案している．この方法では，1 つのアプリケーションによる描画の終了がシステムで発生するすべての描画処理の終了と一致している必要がある．スマートフォンでは，複数アプリケーションの描画処理が 1 つの描画期間に混在して発生するため，あるアプリケーションの描画終了をとらえて全描画処理の終了と見なすことができない．文献 [20] の方法を適用するためには，1 描画期間の描画処理がすべて終了したことを識別する方法が必要である．本研究の提案方式では，全描画処理の終了したことを推定して GPU を制御する．

3. 提案手法

本研究の目的は，Android において 1 フレーム分の描画を行う描画期間に発生する描画がすべて終了したことを推定し，GPU を現在より消費電力が少ない状態に設定することである．そのため，描画終了を高い確率で推定することが第 1 の課題である．また，提案方式は 1 回の描画期間の全描画処理の終了を推定するため，描画処理が未終了であるのに，すべての描画処理が終了したと誤認識した場合に，消費電力の増加や，描画が 1 回の描画期間内に終了しなくなるというリスク（詳細は 3.1 節で述べる）が発生する．そのため，描画終了を誤認識した場合に発生するリスクを軽減することが第 2 の課題となる．

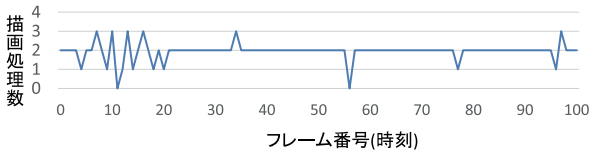


図 14 描画処理数の変化の例
Fig. 14 Transition of drawing number.

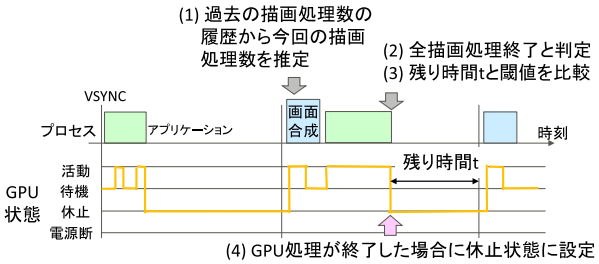


図 15 GPU 状態制御概要
Fig. 15 An example of GPU state transition in proposed method.

3.1 描画処理終了の推定による GPU の状態制御手法

ベンチマークや一部のゲームを除く多くのアプリケーションは，Android フレームワークの画面合成機能が終了時に発信する通知を受けて描画を開始する．これらのアプリケーションは通知を受信すると，描画期間の間に仮想画面を更新し，OpenGL の描画終了機能（eglSwapBuffers）を呼び出す．

そのため，アプリケーションが更新する仮想画面の数はステータスバーにある，電池のインジケータや電波強度，時刻表示といった情報が更新されるタイミング，アプリケーションの画面の構成が大幅に変更される場合，ユーザによる画面操作の変化時を除くと一定となる．実際に仮想画面の描画処理数を 2.4 節と同じアプリケーションの動作ログで確認したところ，アプリケーションの起動や終了時を除き，あまり変化していない（**図 14** はその一例をグラフ化したもの）．

本研究の提案手法では，この性質を利用して 1 描画期間における全描画処理が終了したことを判定する（**図 15**）．1 つの描画期間に実行された仮想画面の描画処理の数（以後，描画処理数と呼ぶ）の履歴をカーネル内の記憶領域に蓄積しておき，新しい描画処理期間が始まったタイミングで，履歴を用いて実行される描画処理数を推定する（図 15 の (1)）．さらに，仮想画面に対する描画終了 API の呼び出しを回数を数えて，全描画処理が終了したか否かを判定する (2)．描画がすべて終了したと判断した際に，描画期間の残り時間 t と閾値を比較し，残り時間が閾値以上の場合のみ，GPU 状態を「休止」その他の場合は「待機」と判定する (3)．GPU の処理が終了した時点で先ほど判定した状態に GPU を設定する (4)．

ただし，推定した描画処理数と実際の描画処理数が一致しない場合がある．推定した描画処理数より実際の描画処理数が大きい場合は，リスクが生じる（**図 16**）．提案方式では，推定した数の処理が終了すると，GPU を休止状態に設定する．しかし，休止状態に入った後に，新たな描画処理が発生すると，以後，休止状態と活動状態の間の遷移が発生する．この状態遷移により，消費電力の増加や，描画処理が遅延する可能性がある．

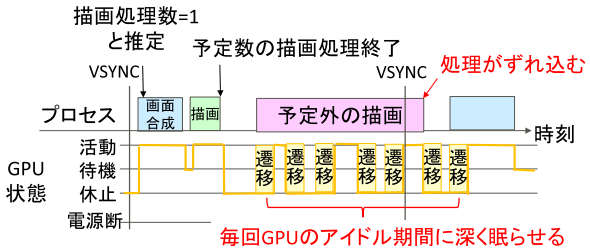


図 16 性能上のリスクのある場合
Fig. 16 The risk of propsed method.

一方，推定値が実際の描画処理数より大きかった場合，GPU は活動状態と待機状態の間で遷移するため，従来手法と同じ状態遷移となり，リスクは発生しない．また，推定値と描画処理数が一致している場合は休止状態に遷移した後は，次の描画期間まで描画処理は発生しないため，リスクは発生しない．

そのため，描画処理数を推定するアルゴリズムの選択には，推定と実際の描画処理数が一致する確率だけでなく，リスクの発生確率にも配慮した．また，リスクを軽減する措置も導入した．これらの詳細は，次節以降で説明する．

3.2 描画処理数推定法

本提案手法では，アプリケーションレベルの描画処理数の履歴から，次の描画期間に実行される描画処理数を推定する．図 14 のように，ユーザがあるアプリを使っている期間において，描画処理数の変化の頻度は低いため，出現頻度で決める方法が考えられるが，利用するアプリケーションを変更した場合やユーザが画面を操作する指の動きが変化した場合等，状況変化への追従に時間がかかることと，統計的な偏りが明確になるほど多くのログを蓄積する必要があるため，必要なログのデータ量や計算量も大きくなる．そのため，比較的短い期間のログを用いて描画処理数を推定するアルゴリズムとして，「(1) 直前の値をそのまま用いる」，「(2) 一定期間同じ処理数が続いた場合にその値を採用する」，「(3) 平均値を求める」が考えられる．

定量的な基準で，最も適したアルゴリズム選択するため，2.4 節と同じアプリケーションのログに対して，上記の 3 種類のアルゴリズムのバリエーション（利用するログの長さ，小数点以下の取扱いを変化させた 42 種類）を適用するシミュレーションを実施し，推定が正しかった確率（的中率）とリスクの発生確率を求めた．

なお，候補としたアルゴリズムは以下のとおり．

- 直前の描画期間の処理数を利用：1 種類
- 過去 n 回の描画期間の処理数が同じであった場合に，推定値を変更：n が 2〜6 の 5 種類
- 過去 n 回の描画処理数の平均値に対して小数点以下切り捨て，切り上げ，もしくは四捨五入を適用：n が 2，3，5，10，15，20 の 6 通り × 3 = 18 種類
- 過去 n 回の描画処理数の線形移動加重平均値に対して小数点以下切り捨て，切り上げ，もしくは四捨五入を適用：n が 2，3，5，10，15，20 の 6 通り × 3 = 18 種類

シミュレーションのプログラムはログ上の VSYNC 間に発生した仮想画面の描画終了の個数（描画処理数）を数え次の描画期間に発生する描画処理数を予想するという，実際に実装する場合と同じ処理を行うプログラムを sed, awk, perl のプログラムの組合せで実現し，推定値と実際に発生した描画数が一致したか否か，リスク（推定値 < 実際の描画処理数）が発生したか否かを集計した．

ただし，候補のアルゴリズムが多いため，シミュレーションの結果に対して，次のような式 (2) で計算される指標を用いて候補を絞り込んだ後，絞り込んだ候補の個々のデータを見て最終的なアルゴリズムを決定した．式 (2) を用いた理由は，推定値と実際の描画処理数が一致する確率（的中率）は高いほうが良いが，リスクの発生確率は抑制したいことと，1 つのアルゴリズムでより多くのアプリケーションに対応するため，アプリケーションの種類による性能差が少ない方が良いためである．

$$\text{得点} = \text{的中率} \times (100 - \text{リスク発生確率}) \div 100 \quad (1)$$

$$\text{指標} = \text{Average(得点)} - \{\text{Max(得点)-Min(得点)}\} \quad (2)$$

42 通りのアルゴリズムの中から，指標の値が高かったもの 8 種類における，的中率の最小値と，リスクの発生確率の最大値を**表 1** に示す．8 つのアルゴリズムのうち，リスクの発生確率が最小で，的中率が最大となる「過去 3 回の描画処理数の単純平均値を求め，小数点以下を切り上げ」（表 1 における 4）を選択した．

3.3 リスク軽減方法

3.1 節で説明したように，推定した描画処理数より実行された描画処理数が多い場合，リスクが発生する．電池残量や電波の受信状況の変化で画面のステータスバーが変化するタイミング等，低い頻度でしか発生しない描画処理数の変化の場合は，当該の描画期間内で対処を行う．具体的には，GPU を休止状態に設定した後に，次の描画期間開始（VSYNC の発生）前に GPU 処理が発生した場合は，処理終了時に GPU を「待機状態」に設定する（**図 17**）．

さらに，アプリケーションの切替え，ユーザの操作にともないアプリケーションの状態が変化した場合等は，その変化の途上で，描画処理数が頻繁に変動する．その際，推定的中率は低下し，リスク発生確率が増加する．このようなリスクには，描画処理数が変動していることを検出し，変動している期間は GPU を休止状態に設定しない（**図 18**）．なお，変動の検出アルゴリズムは，2.4 節と同じアプリケーションの動作ログに対してシミュレーションを行い選択した．シミュレーションのプログラムは 3.2 節のシミュレーションプログラムを拡張し，変動開始と終了の検出を行う機能を追加した．以下の項でシミュレーション結果の詳細について述べる．

表 1 描画処理数の推定アルゴリズムの各性能値
Table 1 Performance of drawing number estimation in proposed method.

番号	算出方法	小数点以下	期間	的中率最小値 (%)	リスク最大値 (%)
1	同一描画	—	2	77.1	9.7
2	処理数連続		3	76.5	9.7
3	単純平均	切り上げ	2	76.0	7.5
4			3	76.3	6.5
5		四捨五入	2	76.0	7.5
6			3	75.5	8.4
7	加重移動平均	切り上げ	2	76.0	7.4
8			3	74.9	6.5

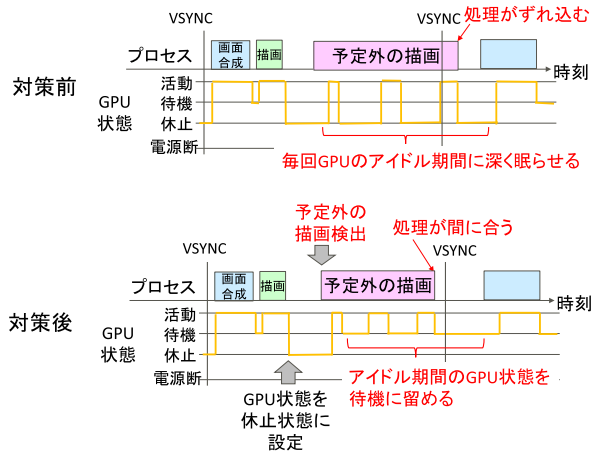


図 17 描画遅れリスクの軽減手法
Fig. 17 Risk avoidance method.

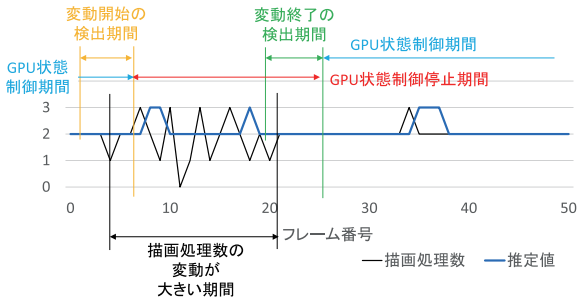


図 18 変動の検出
Fig. 18 Drawing number fluctuation detection.

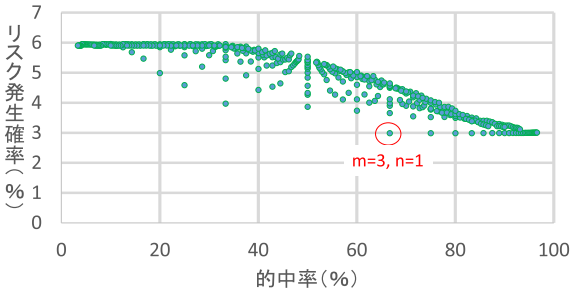


図 19 推定の的中率とリスクの発生確率の関係
Fig. 19 Drawing number estimation and probability of risk.

3.3.1 変動開始の検出

アプリケーションの切替え等により発生する，高頻度の描画処理数の変動を検出する際に，実際の描画処理数に基づいて判定する方法と，描画処理数の推定が的中した率に基づいて判定する方法の 2 種類がある．本研究の手法では，過去の描画処理数の履歴に基づいて次の描画期間の処理数を推定することから，変動の検出も推定が的中する確率で行う．

変動の開始を検出するため，過去 n 回のうち，推定が m 回外れたことで変動の開始とする場合について n を 3 から 30，m を 1 から $n-1$ まで変化させ，的中率と，リスクの発生確率のシミュレーション結果を**図 19** に示す．本研究の目的に最も適しているのは，原点に近い点（リスクが低く，短い期間で判定できる）に相当する方法である．このシミュレーション結果から，過去 3 回の推定結果のうち，1 回推定が外れた場合（$n=3$，$m=1$）で判定する方法を選択した．

3.3.2 変動終了の検出

変動終了を判定するには，変動開始判定より高い確率で推定が成功する必要がある．また，消費電力の削減効果を大きくするため，変動終了を早く検出する必要がある．前項の結果で，過去 3 回の推定のうち 1 回の推定外れで変動開始と判定することから，推定が 2 回もしくは 3 回連続して的中した場合に，変動の終了と判定する方法を候補として前項と同じく取得したログに対してリスクの発生確率を計算するシミュレーションを行った．

推定が 2 回もしくは 3 回連続して的中した場合に変動の終了と判定する方法についてリスクの発生確率の差は 0.2%以下であった．そのため，2 回連続で推定が的中した場合に変動終了と判定し，GPU の状態制御を行うアルゴリズムを採用した．

3.4 描画処理の終了が検出できない場合への対応

本提案手法では，アプリケーションレベルの描画処理の終了を検出するため，OpenGL の特定機能の呼び出しを観測する必要がある．しかし，OpenGL のライブラリはバイナリ提供の場合もあり，この場合は変更を加えることができない．一方，Android フレームワークを用いて描画を行うアプリケーションは，フレームワークの OpenGL 機能経由で，OpenGL ライブラリを呼び出す．以上のような理由から，本提案手法では，アプリケーションの描画処理終了のイベントをフレームワークで検出する．

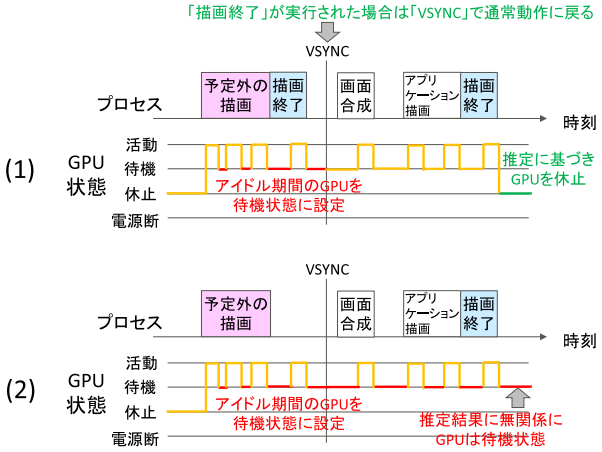


図 20 例外対応手法
Fig. 20 The risk avoidance method.

ただし，Android のアプリケーションは NDK [21] を利用することにより，フレームワークを迂回し，直接 OpenGL のライブラリを操作するアプリケーションを作成することが可能である．これらのアプリケーションの実行時は，描画終了を示すイベントが一度も観測されないため，描画処理数の推定より実際の描画処理数がつねに大きくなり，リスクが連続して発生する．

図 20 において，(1) は通常のアプリケーション実行時かつ，「推定値 < 実際の処理数」の場合．(2) はフレームワークを迂回して描画を行うアプリケーションの場合である．通常のアプリケーションで描画処理数の推定が外れた場合は，「予定外の描画」の末尾に「仮想画面に対する描画終了」のイベントが検出される．それに対して，フレームワークを迂回して描画を行うアプリケーションは「予定外の描画」の後で「描画終了」のイベントが観測されないまま，次の描画期間が開始される．

この現象を利用して，「予定外の描画」から「VSYNC」までの間に「描画終了」のイベントが観測されるか否かでどちらの状況かを判定する．もし，(1) の場合は，新しい描画期間の開始から通常の動作に戻るが，(2) の場合は描画期間数回の間，GPU を休止状態に設定する処理は行わない．なお，評価に用いたプロトタイプでは，描画期間 4 回の間 GPU を休止状態に設定する動作を行わない．

4. 評価

本研究の提案手法では，描画期間 1 回に実行される描画処理数の統計的な性質に基づいて全描画終了を推定し，描画期間の残り時間と閾値を比較して GPU 状態を判断する．

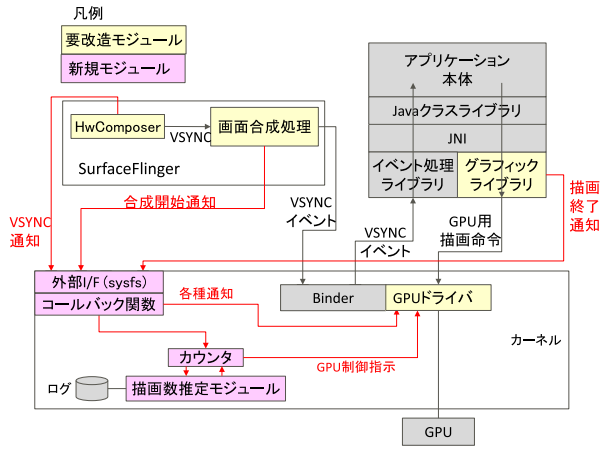


図 21 システム構成図
Fig. 21 Prototype system.

この手法の閾値は SoC の種類，クロック発生回路や電源回路等の構成により異なる．

そのため本評価では，描画処理数の推定が実際の描画処理数と一致（的中）する確率のほか，現在の端末で提案手法が有効であるか否かを判定するため，以下の項目について評価した．

- 一般的な端末において，GPU のアイドル期間がどの程度連続すれば消費電力が削減可能であるか．
- 主要なアプリケーションで電力が削減できるほど長期間 GPU のアイドル期間が存在するか否か．
- 主要なアプリケーションで実際にどの程度の消費電力が削減できるか．

今回の評価用に Nexus5 で動作するプロトタイプを開発した．評価に用いたアプリケーションは 2.4 節と同じものを用い，同様の操作を行った．

4.1 プロトタイプ

本評価用のプロトタイプは，Nexus5 用の Android4.4.2 を拡張して，提案手法を実装した（**図 21**）．描画期間の開始やアプリケーションレベルの描画処理終了を検出するため，Android フレームワークの OpenGL 対応部分，画面合成機能（SurfaceFlinger）を拡張し，検出したイベントをデバイスドライバに伝えるため，アプリケーションが読み書き可能なカーネル内の記憶領域（sysfs）に書き込む．カーネルの GPU ドライバでは，イベントの通知（sysfs への書き込み）を契機として，描画処理数の履歴の蓄積，描画処理数推定値の算出，描画処理数変動の検出を行うとともに，推定値と同じ数の描画処理が終了した場合に GPU の状態を制御する．

4.2 描画処理数の推定が的中する確率

表 2 は描画処理数推定の的中率を示している．この的中率は，アプリケーション利用期間のうち，描画処理数が

表 2 推定的中率
Table 2 Drawing number estimation probability.

アプリケーション名	推定的中率 (%)
Android ブラウザ	85.5
Chrome ブラウザ	81.5
設定画面	94.8
youtube	98.6

変動していないと判定された期間における描画処理数の推定値と実際の描画処理数が一致した確率である．この評価には，プロトタイプのカーネルに描画処理数の推定が正しかったか否かをログに出力する機能を持たせ，実際のアプリケーションの操作時のログから算出した．最も低いブラウザでも 80%以上の的中率となり，端末設定画面や youtube では 90%を超えており，推定により描画処理の終了を判定する方法の有効性は認められる．

4.3 Nexus5 の GPU 状態制御の閾値

提案手法では，全描画終了時の描画期間の残り時間（アイドル期間の長さで，単位はミリ秒）と閾値（整数）を比較して，GPU の状態を待機状態にするか，休止状態にするか判定する．この閾値は，GPU 状態を休止状態にした場合と，待機状態にした場合で消費電力が同じとなる値（損益分岐値）にする．アイドル期間が損益分岐値より長い場合，GPU を休止状態にすることにより，消費電力を削減する（益を得る）ことが可能であるが，アイドル期間が損益分岐値より短い場合，休止状態に設定することにより，消費電力が従来より増加する（電力的に損をする）．

閾値が損益分岐値より大きい場合，GPU が休止状態に設定される確率が減り，提案方式による消費電力削減効果も小さくなる．同様に，閾値が損益分岐値より小さい場合は，GPU を休止状態にすることにより電力ロスが発生し，消費電力の削減効果が減少する．つまり，閾値を損益分岐値と同じにした場合に，消費電力の削減効果が最大（端末の消費電力は最小）となる．

このことから，実際のアプリケーションを動作させた端末の消費電力をさまざまな閾値で測定し，消費電力が最小となる閾値を求めた．この際，アプリケーションは人の操作が介在せず，さまざまな長さのアイドル期間を発生させる youtube を用い，端末の電源回路を改造し，安定化電源から端末に供給された電力を測定した．

図 22 は測定結果である．閾値を 8 ms とした場合に，端末の消費電力が最小（提案手法の効果が最大）となったことから，以後の評価における閾値は 8 ms とした．

4.4 電力削減効果が見込めるアプリケーション

次に，GPU の状態制御の閾値と GPU が休止状態に設定される時間の関係のデータを収集した（**図 23**）．たとえ

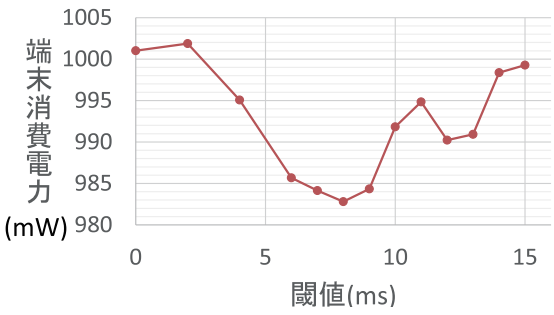


図 22 損益分岐値
Fig. 22 Break-even point of the threshold in proposed method.

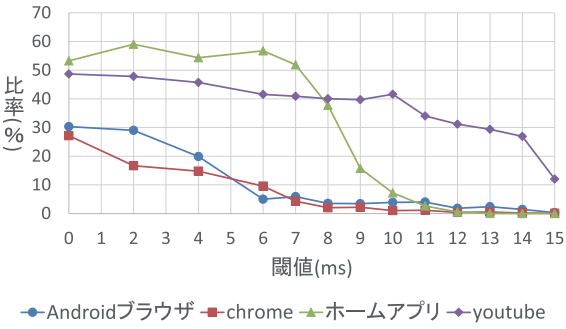


図 23 閾値と sleep 時間比率グラフ
Fig. 23 Threshold and GPU sleep time.

ば，閾値が 4 の場合は描画終了から次の描画期間の開始までに 4 ms 以上存在した場合に GPU を休止状態にする．そのとき，設定画面と youtube は，アプリケーション利用時間のほぼ半分程度の時間 GPU が休止状態（低電力状態）となり，2 種類ブラウザは約 20%の時間，GPU が休止状態となることを示している．

一方，4.3 節の評価結果から，Nexus5 は閾値は 8 ms にした場合に効果が最大となる．また，図 23 から分かるように，2 種類のブラウザは閾値が 8 ms では，GPU が休止状態となる可能性が低く，消費電力の削減効果を見込むことはできない．それに対して，ホームアプリや youtube では，アプリケーション動作時間のうち，多くの時間 GPU を休止状態に設定できるため，消費電力の削減効果が期待できる．

4.5 消費電力の削減効果

前節の結果から，youtube およびホームアプリでは，提案手法の効果が見込めるため，この 2 種類について端末の消費電力削減効果を測定した．

なお，Nexus5 に搭載された GPU の場合，待機状態と休止状態の場合の端末全体の消費電力の差は GPU のパワーレベル（周波数と電圧の組合せ）によって異なる．Nexus5 の場合 7 段階のパワーレベルが存在し，パワーレベル最低の場合で 64 mW，パワーレベル最高の場合は 402 mW の差となる．

一般的に，CPU や GPU の周波数は CPU や GPU の稼働率で変化するが，Android 端末の GPU パワーレベルは GPU の稼働率に加えて，ユーザが画面に触れたか否かで変化する．CPU や GPU の処理発生から，周波数が増加するまでに数十 ms 必要であるため，周波数が低くなっている状態で画面にタッチした場合にユーザがレスポンスの悪さを感じてしまう．これを避けるために，画面タッチ後しばらくの間 CPU と GPU の動作周波数を増加させる（動作周波数の最低値を引き上げる）機能が実装されているためである．

表 3 各アプリケーションにおける消費電力の削減効果
Table 3 Power reduction in Android applications.

アプリケーション名	電力削減値 (mW)	端末の全消費電力に対する比率 (%)
youtube	17.6	1.8
スクロール	63.2	4.7
スワイプ	30.4	2.0

このため，ホームアプリについては画面をスクロールする場合とスワイプの場合で分けて消費電力を測定した（**表 3**）．最も効果が低い youtube で端末消費電力の 1.8%，効果が最大のホームアプリでスクロールする場合で 4.7%削減することができた．

評価対象のアプリケーション動作時の端末のパワーレベルを確認したところ，youtube の動画再生中は画面に触れることがないため，最低のパワーレベルで動作する．また．youtube 以外のアプリケーションにおいて，画面に触れた直後のみ GPU のパワーレベルが増加するが，それ以外の期間は，最低のパワーレベルで動作していた．

youtube の場合，動作時間の約 40%が休止状態となり，かつ，その期間はパワーレベルが最小であるため，状態遷移のロスを無視した場合で，64 mW × 0.4 = 25.6 mW の削減効果，実際の削減効果が 17.6 mW であり，妥当な値と考えられる．youtube 以外のアプリケーションは人間による画面操作が行われることによるパワーレベルの増加が発生するため，youtube より大きく消費電力を削減することができている．ホームアプリの場合，スクロールとスワイプで差があるのは，画面操作中，指が触れている時間が違い，GPU のパワーレベルを増加させている期間が異なるためである．

5. まとめ

Android を搭載した端末における消費電力を削減することを狙い，描画処理終了後の GPU のアイドル期間の状態を制御するため，アプリケーションの描画処理数を推定し，該当数の描画処理が終了した場合に GPU を休止状態に設定する手法を提案し，提案手法のプロトタイプを Nexus5 で動作させて性能評価を行った．

性能評価では，ある描画期間に実行される描画処理数の

推定が実際の処理数と一致した確率，実際に利用されている端末で，状態制御で消費電力を削減するためには，どの程度アイドル期間が続く必要があるか，主要なアプリケーションで状態制御による消費電力削減が可能なほど長いアイドル期間が存在するか否か，消費電力削減が可能なアプリケーションでどの程度端末の消費電力が削減できるかの4点について評価を行った．

評価の結果，ブラウザでは約80%，その他のアプリケーションでは90%以上の確率で描画処理数の推定値が実際の描画処理数と一致した．また，Nexus5ではGPUの状態制御で電力を最小にするためには，描画終了後のアイドル期間が少なくとも8ms以上必要であることが分かった．ホームアプリやyoutubeは8ms以上のアイドル期間が比較的頻繁に発生するため，電力削減効果があるが，ブラウザでは長いアイドル期間がほとんど発生せず，GPUの状態制御による電力削減は困難である．

効果が見込めるアプリケーションについて実際の端末の消費電力の削減効果を測定したところ，youtubeで1.8%，ホームアプリでスワイプする場合に2.0%，スクロールする場合で4.7%，本提案方式で端末の消費電力を削減した．

参考文献

[1] Google: Android Developer, Google (online), available from ⟨http://developer.android.com/⟩ (accessed 2015-08-07).

[2] 日経トレンディネット：大容量は必須，LTEでバッテリーが重要なワケ，日経BP社（オンライン），入手先 ⟨http://trendy.nikkeibp.co.jp/article/column/20111212/1038972/?P=2&rt=nocnt⟩（参照 2015-08-07）．

[3] MMD研究所：2013年スマートフォン購入者の満足度調査（Android編），MMD研究所 (2013).

[4] Pallipadi, V., Li, S. and Belay, A.: cpuidle: Do nothing, efficiently, *Proceedings of the Linux Symposium*, Vol.2, Citeseer (2007).

[5] Helmbold, D.P., Long, D.D., Sconyers, T.L. and Sherrod, B.: Adaptive disk spin-down for mobile computers, *Mobile Networks and Applications*, Vol.5, No.4, pp.285–297 (2000).

[6] Google: Nexus 5, Google (online), available from ⟨http://www.google.co.jp/nexus/5/⟩ (accessed 2015-08-07).

[7] Torvalds, L.: mainline kernel, kernel.org (online), available from ⟨https://www.kernel.org/⟩ (accessed 2015-08-07).

[8] Ubuntu: Ubuntu Kernel, Canonical (online), available from ⟨http://kernel.ubuntu.com/git?p=ubuntu/ubuntu-trusty.git;a=summary⟩ (accessed 2014-03-10).

[9] TI: ti-linux-kernel, Texas Instruments (online), available from ⟨http://git.ti.com/gitweb/?p=ti-linux-kernel/ti-linux-kernel.git;a=shortlog;h=refs/heads/ti-linux-3.14.y⟩ (accessed 2015-08-07).

[10] Google: Kernel source, Google (online), available from ⟨https://android.googlesource.com/kernel/msm.git⟩ (accessed 2015-08-07).

[11] Suda, R. et al.: Power efficient large matrices multiplication by load scheduling on multi-core and GPU platform with CUDA, *International Conference on Computational Science and Engineering, 2009, CSE'09*, Vol.1, pp.424–429, IEEE (2009).

[12] Duato, J., Pena, A.J., Silla, F., Mayo, R. and Quintana-Ortí, E.S.: rCUDA: Reducing the number of GPU-based accelerators in high performance clusters, *2010 International Conference on High Performance Computing and Simulation* (*HPCS*), pp.224–231, IEEE (2010).

[13] 長坂，丸山，額田，遠藤，松岡：GPUにおけるモデルに基づいた電力効率の最適化，計算機アーキテクチャ研究会報告，Vol.2010, No.2, pp.1–6, 情報処理学会 (2010).

[14] Ma, X., Dong, M., Zhong, L. and Deng, Z.: Statistical power consumption analysis and modeling for GPU-based computing, *Proc. ACM SOSP Workshop on Power Aware Computing and Systems* (*HotPower*) (2009).

[15] Collange, S., Defour, D. and Tisserand, A.: Power consumption of gpus from a software perspective, *Computational Science–ICCS 2009*, pp.914–923, Springer (2009).

[16] Zhang, Y., Hu, Y., Li, B. and Peng, L.: Performance and power analysis of ATI GPU: A statistical approach, *2011 6th IEEE International Conference on Networking, Architecture and Storage* (*NAS*), pp.149–158, IEEE (2011).

[17] Jiao, Y., Lin, H., Balaji, P. and Feng, W.: Power and performance characterization of computational kernels on the gpu, *Green Computing and Communications* (*GreenCom*), *2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing* (*CPSCom*), pp.221–228, IEEE (2010).

[18] Nagasaka, H., Maruyama, N., Nukada, A., Endo, T. and Matsuoka, S.: Statistical power modeling of GPU kernels using performance counters, *Green Computing Conference, 2010 International*, pp.115–122, IEEE (2010).

[19] NTTドコモ：実使用時間について，NTTドコモ（オンライン），入手先 ⟨https://www.nttdocomo.co.jp/product/battery_life/⟩（参照 2015-08-07）．

[20] 大場：回路動作制御装置および情報処理装置，特開 2005-62798 (2005).

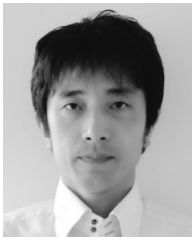
[21] Google: Android NDK, Google (online), available from ⟨https://developer.android.com/tools/sdk/ndk/index.html⟩ (accessed 2015-08-07).

野呂 正明 （正会員）

1966年生．1988年名古屋大学工学部電気工学科卒業．1990年同大学大学院情報工学研究科博士課程前期課程修了．同年株式会社富士通研究所入社．2002年よりTAOおよびNICTに出向．2008年大阪大学大学院情報科学研究科博士課程後期課程修了．2008年より現職．ネットワークの品質制御，データ転送プロトコル，OSの研究開発に従事．2014年情報処理学会山下記念研究賞受賞．

村上 岳生 （正会員）

1968 年生．1991 年東京大学理学部情報科学科卒業．同年（株）富士通研究所入社．システムソフトウェアの研究に従事．



上和田 徹 （正会員）

1996 年大阪大学大学院基礎工学研究科物理系情報工学専攻修士課程修了．同年富士通（株）入社．（株）富士通研究所ユビキタスシステム研究所ユビキタスデバイスプロジェクト主任研究員．PC およびスマートフォン等の端末向けソフトウェアプラットフォームの研究に従事．

石原 輝雄

1987 年東北大学工学部通信工学科卒業．同年富士通（株）入社．通信装置向け LSI 設計開発に従事．1997 年から（株）富士通研究所にて，携帯端末向けデバイスの研究開発を行う．現在，メンテナンスフリーなセンシングデバイスの研究開発に従事．