

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

概要：スマートフォンに関する市場調査の結果では 40%以上のユーザが機種選定時に重視する項目に「電池持ち」を挙げており、GPU は端末において消費電力の大きな要素の 1 つである。Android 端末の GPU は、1 回の画面更新の期間に、頻繁にアクティブとアイドル状態を行き来しているが、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 フレームの描画の終了後に長いアイドル期間が現れる確率が高い。そのため、1 フレーム分の描画の終了を把握することができれば、GPU を低電力状態に設定することが可能となる。本研究では、1 フレームにおけるアプリレベルの描画処理数の履歴から、次のフレームのアプリレベルの描画処理数を推定し、その数の描画処理が終了した時点で GPU の状態を制御する。

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、youtube、ホーム画面、端末設定画面でそれぞれ 4.4mA、7.6mA、15.8mA の端末の消費電流を削減できた。

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

1. 背景

Android[1] ^{*1} を搭載した端末を始めとして、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、画面サイズの拡大、高速ではあるが、単位時間あたりの消費電力が大きな無線技術の普及など、物理的に端末の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要求は以前から高く、市場調査の結果 [2] でも端末の選定時に電池の持続時間を重視するユーザの比率が 40%を超える状況であり、消費電力の削減はいまだ重要な課題である。

スマートフォンにおいてディスプレイ、無線、CPU や GPU といった演算部分の 3 つが消費電力の大きな要素である。このうち、CPU は処理がない「アイドル期間」にクロックや電源の供給を止めるといった低電力状態に設定することで消費電力を抑制している。

一般に、GPU や CPU で状態を変更した場合、状態変更時に通常より多くの電流が流れる (図 1)。図の例では、(1) の期間では電源やクロックの供給回路に状態遷移を起こさせるため、(2) の期間では、電源やクロック供給回路の状態遷移の電力に加えて、回路全体が動作するために電荷を

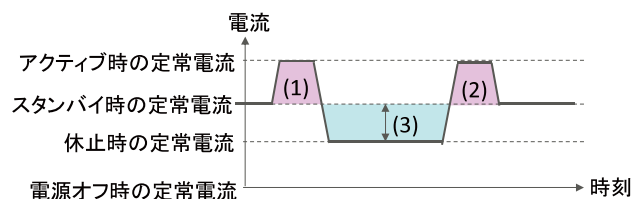


図 1 状態制御と消費電流

溜める必要があるためである。また、 $(1) + (2) < (3)$ でなければ消費電力は減少しないため、低電力状態に設定する時間が短い場合、逆に消費電力が増加する。実際に状態遷移で消費電力を削減するためには、CPU や GPU での処理が長時間発生しない場合にかぎり、状態を変更する必要がある。CPU では消費電力や状態変更によるロスが異なる多くの状態が利用できるため、アイドル期間の長さに応じて状態を使い分けている。GPU も種類は少ないが、アイドル期間に設定できる状態があり、一定以上の時間連続して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。

Android の画面は複数の仮想画面を合成することで成り立っており、各アプリは 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書いた上で、描画終了をシステムに通知することで、次の画面合成時に描画済みの画像が実際の画面に反映される。Android は最大で秒 60 フレームの更新を行うため、1 フレームは約 16.6ms となる。実際の描画処理は、図 2 のような 3 段階のパイプ

¹ 富士通研究所
FUJITSU LABORATORIES LTD 4-1-1 Kamikodanaka,
Nakahara-ku, Kawasaki, Kanagawa, Japan

^{a)} noro@jp.fujitsu.com

^{*1} Android は google の登録商標です。

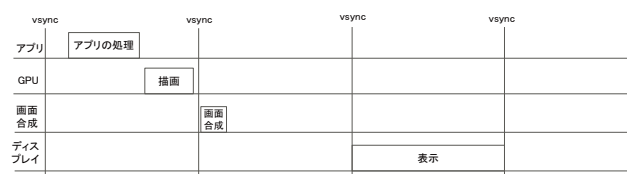


図2 描画の3段階パイプライン

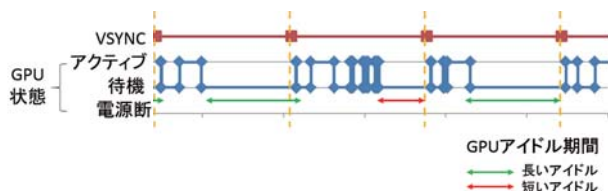


図3 描画の例

ラインで構成されており、1番目のフレームで描画された結果が2番目のフレームの先頭で実際の画面に出力される画像に合成され、3番目のフレームで合成済み画面が実際のディスプレイに表示される。この際、画面の合成は前のフレームの描画の終了のタイミングに無関係に、2番目のフレームの先頭で実行される。このため、1番目のフレームでは任意の数のアプリが任意のタイミングで任意の回数GPUを利用することが可能であり、あるGPUの処理が終了し、処理が無い状態になった場合に、次回のGPU処理が発生するまでの時間を判定することはできない。そのため、現在のスマートフォンでは、GPUの処理が終わりアイドル期間に入る場合、後続の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、状態遷移ロスが少ないが、比較的消費電力の大きな状態（図3における「待機」状態）に設定している。

図3は、ブラウザでwebページを閲覧した場合の、GPUの動作の例である。この図のように、Androidでは1フレーム内の描画処理が全て終了した後に、比較的長い時間GPUがアイドル状態となる確率が高い。そのため、1フレーム内の全ての描画処理が終了したことを識別できれば、GPUを低電力状態に設定することが可能となる。本研究では、1フレーム内の描画処理終了を描画処理の統計的な性質に基いて推定し、GPU状態を制御する手法を提案する。

CPUの消費電力削減は長い間CPUベンダの主要な課題の一つであったため、電力やクロックを供給する範囲を限定するなどにより、省電力効果は少ないものの、アイドル期間が短い場合でも一定の電力削減が可能な状態から、効果が大きい、長いアイドル期間を必要とする状態まで、消費電力の異なる多くの動作モードを実装してきた。さらに、OS側も利用可能な多くの動作モードを使い分けることで消費電力の低減を実現している。

それに対して、GPUは利用可能な状態が少なく、状態制御で消費電力を削減するために必要なアイドル期間が長い

可能性がある。どの程度の時間省電力モードに設定し続けた場合に消費電力を削減できるかは、SoCの種類、クロック発生回路や電源回路などの構成により異なるため、機種毎に判定する必要がある。

以上のことから、本研究における評価では、提案方式における1フレーム内の描画処理終了の推定が実際の描画処理の終了と一致する確率の他、現在の端末で提案方式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明らかにする。

- 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。

2. 従来のGPUの省電力技術

第1章で説明したように、現在のAndroidでは、GPUにおける処理がない期間はGPUを常に待機状態に設定しているが、多くのアプリでは画面合成の終了を検知して描画を開始する。そのため、1フレーム分の描画処理終了後は、次のフレームの描画処理開始まで、比較的長い時間GPUは利用されない確率が高い。以上のことから、1フレーム分の描画処理が全て終了したことを識別し、次のフレームの開始時刻までの残り時間が判定できれば、GPUを消費電力の少ない状態に設定することで、GPUの消費電力を低減できる可能性がある。

[3]はこの性質を利用し、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捉えて、それを1フレーム分の描画の終了とみなし、GPUを低電力状態に設定する手法を提案しているが、複数アプリ、複数描画処理が1フレーム内に混在するような場合、1つのアプリケーションによる描画終了と1フレーム内の全ての描画処理の終了は一致しないため、Androidには適用できない。

さらに、GPGPUの利用拡大に伴い、GPUの消費電力の解析の研究[4][5][6]、GPUの計算能力と消費電力の関係の研究[7][8][9]、消費電力を意識したGPGPUの方式の研究[10][11]などが行われている。これらの研究は、1つの処理の処理時間、処理の発生時間間隔、処理数等の性質がある程度わかっている場合において、処理をGPUに割り付けるスケジューリングアルゴリズムを工夫することで、消費電力を抑えつつ、計算能力を最大化するための研究であり、スマートフォンにおける描画のように、どの程度の処理がいつ発生するか不明である場合には適用できない。

3. 提案手法

本研究の目的は、Androidにおいて1フレームの期間にアプリが行う描画が全て終了したことを認識し、GPUを

低電力状態に設定することである。しかし、Android には全てのアプリケーションが描画を終了したことを明示的に示すイベントが存在しないため、本研究の提案方式では、画面の一部となる仮想画面に対する描画終了のイベントが発生した数の統計的な情報から、全描画の終了を推定する。そのため、本研究の課題は以下の3種類となる。

- (1) 実際の描画終了を高い確率で推定可能な方法を確定する。
- (2) 描画終了を誤認識した場合に発生する性能上のリスクを軽減する。
- (3) 実行中のアプリの構成上、仮想画面の描画終了を検出できない場合への対応。

3.1 描画処理終了の推定による GPU の状態制御手法

Android のアプリケーションのうち、ベンチマークや一部のゲームなど Android フレームワークに頼らず描画を行う場合を除き、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。これは、Android フレームワークが提供する API の性質によるものである。

さらに、1つの仮想画面に描画可能なアプリのスレッドは1つに限定されており、1つのアプリが複数の仮想画面を利用する場合は、仮想画面数に対応した描画を行うスレッドが存在する。1つのスレッドが GPU を利用する回数はアプリの実装により様々となるが、Android のホーム画面や端末の設定画面のように、Android の標準的な API (View クラス) を用いて作られたアプリやブラウザのようなアプリでは、ユーザが画面を操作している間は1フレームに1回仮想画面を更新し、ユーザが画面を操作していない期間は画面を更新しない。また、1つの仮想画面への書き込み終了時は OpenGL の特定の機能 (eglSwapBuffers) が必ず呼び出されるため、この機能が利用されたことを識別することで、1つの仮想画面に対する描画処理が終了したことは確認できる。

そのため、アプリ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やアプリ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り、描画フレーム毎に頻繁に変化することはない。

この性質は機種や Android のバージョンに関係なく、ベンチマークや一部のゲームを除くアプリケーションで成立し、今後も描画 API が大きく変更されない限り成立する。実際にその変化を端末設定画面など、いくつかのアプリケーションの動作ログ (図4はその一例) で確認したところ、アプリ起動や終了時を除き、あまり変化していないことが確認できた。

本研究の提案方式 (図5) では、1フレーム分の描画処理数の履歴をカーネルに蓄積し、新しいフレームがはじまっ

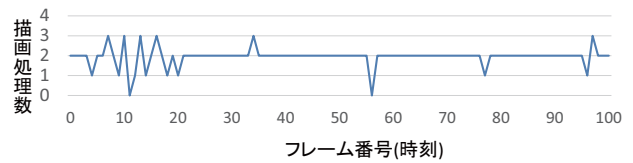


図4 描画処理数の変化の例

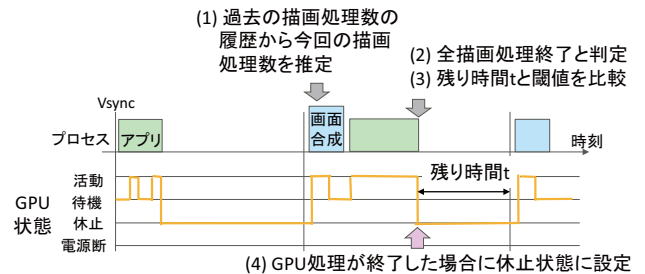


図5 GPU 状態制御概要

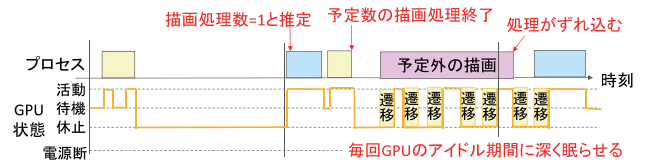


図6 性能上のリスクのあるフレーム

たタイミングで新しいフレームにおける描画処理数 (仮想画面の更新枚数) を推定する (図5の(1)). さらに、1フレーム内における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、1フレーム内の描画が全て終了したか否かを判定する (図の(2)). 1フレーム分の描画が全て終了したと判断した際に、1フレームの残り時間 t と閾値を比較し、残り時間が一定以上ある場合のみ、カーネルの GPU ドライバに次回の GPU アイドル期間に設定する状態を消費電力が少ない休止状態にするよう指示する (図の(3)). GPU ドライバは、GPU からの処理終了割り込みを受信し、GPU がアイドルとなった場合に休止もしくは待機状態に設定する (図の(4)).

ただし、アプリレベルの描画処理数を推定し、それに基づいて GPU を制御する場合、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。このうち、実際の描画処理数が推定値より大きい場合に、性能上のリスクが生じる (図6)。推定した数の仮想画面に対する描画処理数と同じ回数の仮想画面に対する描画終了の API 呼び出しを検出以降、OS のカーネルは GPU から通知される処理終了を検出すると、デバイスドライバにより GPU を消費電力の少ない休止状態に設定する。しかし、推定した数より実際に更新される仮想画面の数が多いと、推定数以上の仮想画面に対する処理中の GPU 状態は、短時間のアイドルであっても、休止状態に設定されるため、休止状態と活動状態の間の遷移が多く発生する。

GPU が活動状態に遷移するためには必要な時間は「待

機状態」と「休止状態」を比較すると周辺回路や GPU に対する電源やクロックの供給の開始や、回路に電荷が溜まる時間が必要となるため、「待機状態」の方が長いだけでなく、状態遷移に必要な電力（図 1 における (2)）も大きい。そのため、休止状態と活動状態間の状態遷移による遅延が複数回発生した場合、1 フレーム内に描画処理が終了しない確率が増加するだけでなく、GPU の消費電力が増加するリスクがある。そのため、描画処理数を推定するアルゴリズムを選択する際には、推定が正しい確率だけでなく、リスクのあるフレームが発生する確率も考慮する必要がある。

さらに、リスクのあるフレームが発生した場合に、性能や電力上の問題を軽減する措置が必要であり、GPU の状態遷移における「休止状態」と「活動状態」間の遷移回数の削減と共に、描画処理終了の推定の的中確率が低下した場合や、仮想画面に対する描画終了を観測できない種類のアプリケーションの実行時に、GPU の休止状態への制御を中止する。これらの個々の方法については次節以降で詳細に説明する。

一方、推定値が実際の描画処理数より大きかった場合、OS は従来と同じく、GPU での処理の有無で活動状態と待機状態の間で遷移させるため、従来と同じ状態遷移となる。この場合、消費電力が削減できない代わりに、リスクも発生しない。また、推定値と描画処理数が一致している場合は、GPU を休止状態に設定した場合に電力が削減できる基準の時間より早く描画が終了した場合のみ GPU を休止状態に設定する上、次のフレーム開始まで描画は発生しないため、リスクは発生しない。

3.2 描画処理数推定

本研究では、アプリレベルの描画処理数の履歴から、次のフレームで実行される描画処理数を推定する必要があるが、具体的なアルゴリズムを決定するため、実際の端末の動作ログ（ホーム画面、chrome ブラウザ）に対して 42 種類の推定手法を適用し、推定が正しかった確率（的中率）とリスクのあるフレームの発生確率を求めた上で適したものを選択した。候補とした推定手法は以下の通り（表 1 はその一覧）。

- 直前のフレームにおける描画処理数のそのまま利用：1 種類
- 過去 n フレームにおける描画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 フレーム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 種類 $\times 3$
- 過去 n フレーム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 種類 $\times 3$

表 1 描画処理数の推定アルゴリズム候補

算出方法	利用ログ量	小数点以下の取り扱い
直前の値を採用	1	—
同一描画処理数連続	2,3,4,5,6	—
単純平均	2,3,5,10,15,20	切り上げ、切り捨て、四捨五入
加重移動平均		

候補が多いため、次のような式（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 を用いた理由としては、推定値実際の描画数が一致する確率（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。そのため、各候補の得点はリスクに応じて的中率から値が小さくなる式とした。また、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、アプリケーション間の差が少ない方が良いため、得点の差が小さいものが良い値となる指標を計算し、指標で候補を絞り込んだ。また、ブラウザとホーム画面を用いた理由としては、両者共にユーザに使用される頻度が高いことと、Android が提供する主要な描画パターン（View クラスによる実装と OpenGL を用いた実装）を網羅することができるためである。

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

$$\text{指標} = \text{全アプリの得点の平均値}$$

$$- \text{得点の最高と最低の差} \quad (2)$$

式における的中率は全フレーム中で描画処理数の推定値が実際に行われた仮想画面に対する処理数と一致した確率（%）、リスク発生確率は全フレームにおいて、リスクのあるフレームが（推定値より実際の描画処理数が多い場合）が発生した確率（%）である。式からわかるように、的中率が高く、リスクが低く、かつ、アプリによる性能差が小さいほど、この指標の値は高くなる。42 通りの手法の中から、指標の値が高かったもの 8 種類を選び出し、的中率の最小値と、リスクのあるフレームの発生確率の最大値を表にまとめたものが表 2 である。8 つの最終候補のうち、リスクのあるフレームの発生確率が最小で、的中率が最大となる「直近の 3 つのフレームにおける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 2 における「算出方法」列が「単純平均」、「小数点以下」列が「切り上げ」、「フレーム」列が「3」の行に該当）を選択した。

3.3 性能上のリスク軽減

推定した描画処理数より実際に実行された描画処理数が多い（性能上のリスクのあるフレーム）場合の描画遅れのリスクを軽減するため、一度深く眠らせた後に、再度 GPU が起こされた場合、その後はアイドルとなっても深く眠らせることをしない処理（図 7）を実施する。

表 2 描画処理数の推定アルゴリズムの各性能値

算出方法	小数点以下	フレーム	的中率 (%)	リスク (%)
同一描画 処理数連続	—	2	77.1	9.7
		3	76.5	9.7
単純平均	切り上げ	2	76.0	7.5
		3	76.3	6.5
	四捨五入	2	76.0	7.5
		3	75.5	8.4
加重移動平均	切り上げ	2	76.0	7.4
		3	74.9	6.5

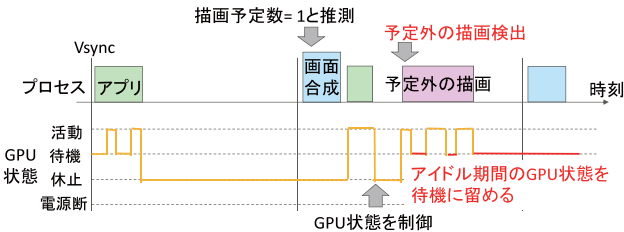


図 7 描画遅れリスクの軽減手法

提案手法では、描画処理数の予測値と同じ回数の描画が行われると、GPU のドライバは次にアイドルになった時点で GPU を休止状態に設定する。この動作をのドライバで記憶しておき、休止状態にした後に新たなフレームがはじまる前に GPU に対するアクセスが発生した場合、ドライバは現在のフレームがリスクのあるフレームであると判定することができる。

さらに、アプリの切替、ユーザの操作に伴いアプリの状態が変化した場合などは、描画処理数が異なる状態に変化する可能性があるが、その変化の途上では、描画処理数が頻繁に変動する可能性がある。その際は、推定的中率は大きく低下し、リスクのあるフレームが出現する確率が増加する。これに対応するため、状態変化途上にあることを検出し、状態が変化している期間は GPU を休止状態に設定しない（図 8 はそのような状態制御の例）ことが、リスク低減の意味で有効である。描画処理数の推定と同様に、実機のログに対して各種の変動検出のアルゴリズムを実装したシミュレーションプログラムを適用し、推定の的中率とリスクの発生確率を用いて具体的な変動検出のアルゴリズムを次の 2 つに確定した。

変動開始の検出 3 フレームのうち、1 フレーム推定が外れた場合に変動開始と判定

変動終了の検出 2 フレーム連続で推定が的中した場合に変動終了と判定

3.3.1 変動開始の検出

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いて次のフレームの描画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

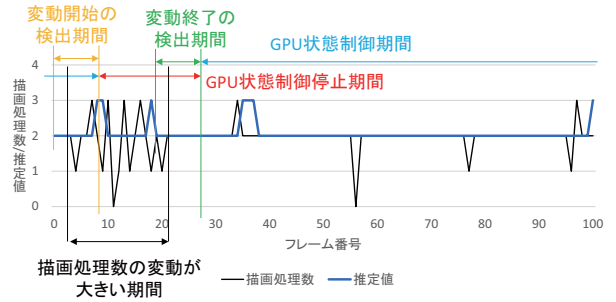


図 8 変動の検出

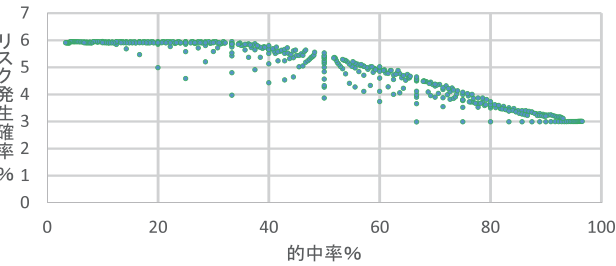


図 9 推定の的中率とリスクの発生確率の関係

で行うこととした。

変動の開始を検出するため、過去 n フレームのうち、推定が m 回外れたことで変動の開始を検出する場合について n を 3~30、 m を 1 から $n-1$ まで変化させ、的中率と、その手法で変動を検出した場合のリスクのあるフレームの発生確率を、実機のログを用いたシミュレーションで求め、それをグラフ化したものが図 9 である。このシミュレーション結果から、リスクが最も低くなるのは、 n の値に無関係に $m=1$ の場合であった。以上から、本研究のプロトタイプでは、 $n=3$ 、 $m=1$ とした。

3.3.2 変動終了の検出

変動の終了を判定するためには、変動開始の判定より推定が高い確率で成功する必要がある。前節の結果では、リスクの削減を重視する場合、変動開始が n フレームのうち、1 フレーム推定が外れた場合で変動開始と判定することが有利であることがわかっており、 $n=3$ を変動の開始の検出方法として選択したことから、描画処理数の推定が 2 もしくは 3 フレーム連続して実際の描画処理数と一致した（推定が的中した）場合に、変動の終了と判定することとした。

さらに、連続して推定が的中するフレーム数で特性が変化するか否か確認するため、ブラウザと設定画面の動作ログを利用したシミュレーションを行なったところ、リスクのあるフレームの発生確率はほぼ同じであるが、GPU の状態制御の対象となるフレーム数が 2 フレームで変動終了と判定するアルゴリズムの方が有利であった。

3.4 描画処理の終了が検出できない場合への対応

本研究では、アプリレベルの描画処理の終了を検出する

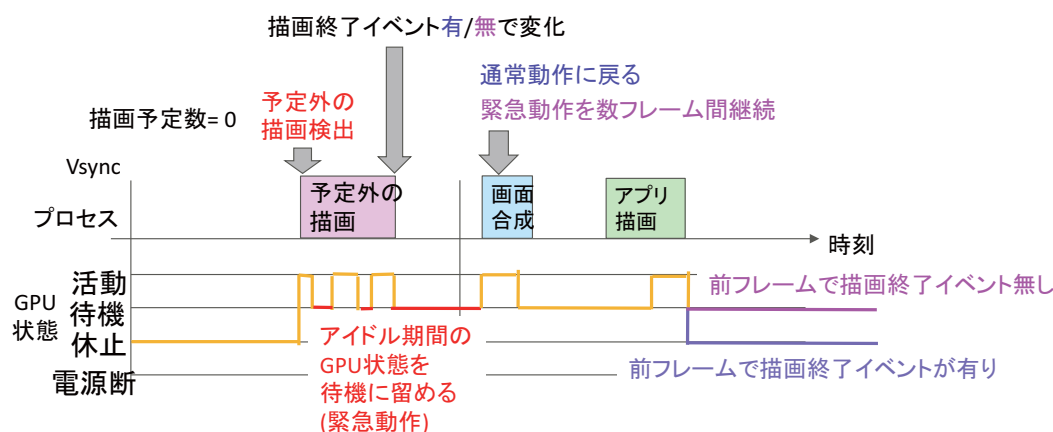


図 10 例外対応手法

ために、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 のライブラリはチップベンダからのバイナリ提供となっている場合もあり、そのような場合はアプリレベルの描画終了の API が利用されたことを外部に通知する拡張を加えることができない。しかし、Android ではフレームワークのネイティブ層に同じ名前の機能が存在し、フレームワーク経由で描画を行うアプリは、このフレームワーク機能を経由して、OpenGL ライブラリの機能を利用している。このことから、フレームワーク側の機能を拡張することで、フレームワーク経由で描画を行うアプリの描画処理終了は検出可能である。以上のような理由から、本研究のプロトタイプでは、描画終了のイベント検出をフレームワークで行うこととした。

ただし、Android のアプリは NDK[12] 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を作成することが可能である。ベンチマークやグラフィックを多用するゲームの一部にはこのような実装を行なっているものが見受けられる。これらのアプリの実行時は、描画処理がフレームワークを経由しない。そのため、本研究のプロトタイプで採用した実装方式では、描画処理の終了を検出できない。すると、この種のアプリが動作している期間中は、描画終了を示すイベントが一度も観測されず、描画処理数を常に 0 と推定する。しかし、実際には描画が発生することから、描画が遅延するリスクの有るフレームが連続して発生することになる。このようなリスクが連続して発生することを避けるためには、この種のアプリが動作している期間中は GPU の状態制御を停止する（GPU を休止状態にしない）ことが望ましい。

NDK 経由などで OpenGL ライブラリを直接利用するアプリの検出は、次のような方法で行う（図 10）。このようなアプリが動作している期間、描画終了機能はフレームワークを迂回して呼び出されるため、各フレームで観測される描画処理終了イベントは常に 0 であるにもかかわらず、

GPU を利用した描画が行われる。逆に、通常のアプリ利用時におけるリスクのあるフレームでは、発生した描画が次のフレームまでズレこまないかぎり、フレーム内で描画終了イベントが発生する。

この性質を利用し、リスクのあるフレームの検出からフレーム終了までの間に、アプリの描画処理の終了を示すイベントが発生した場合は、通常のアプリによるリスクのあるフレームとみなし、それ以外の場合は OpenGL を直接呼び出すアプリを実行していると判定する。OpenGL を直接呼び出すアプリを実行していると考えられる期間は GPU を低電力状態に設定する処理を行わず、通常の GPU の状態制御が行われる。この施策により、消費電力の削減は行えなくなるものの、実際の描画フレームレートの低下や動画再生時のフレーム欠落（「コマ落ち」）を避けることができる。

3.5 プロトタイプ

本研究の提案手法が、1 フレームの全描画終了を高い精度で推定し、GPU の状態を制御することが可能であるかを検証すること、一般的な端末において、GPU をどの程度の時間低電力状態に設定すれば、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実際に電力が削減可能であるかを確認するため、プロトタイプを開発した。

プロトタイプでは、Nexus5[13] 用の Android4.4.2 を拡張して、提案手法を実装（図 11）している。フレームの開始やアプリレベルの描画処理の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、sysfs にイベント発生を書き込む。カーネルの GPU ドライバでは、sysfs への書き込みを検出し、1 フレーム開始時に直前のフレームで実行された描画処理数の計算、新しいフレームにおける描画処理数の推定値算出、描画処理数変動の検出を行うと共に、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。

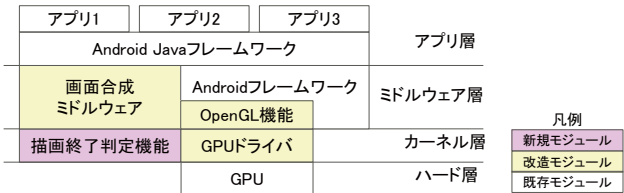


図 11 システム構成図

表 3 推定の中率

アプリ名	推定の中率 (%)
Android ブラウザ	85.5
Chrome ブラウザ	81.5
設定画面	94.8
youtube	98.6

なお、プロトタイプ開発の規模は、Android フレームワーク部分で約 200 行、カーネルのデバイスドライバで約 1300 行であった。

4. 評価

提案手法の性能を評価するため、プロトタイプを Nexus5 で動作させて性能評価を行なった。評価に用いたアプリケーションは youtube, ブラウザ, 端末設定画面やホーム画面である。youtube は横長全画面表示で動画を再生した。またブラウザでは、1 秒に 1 回程度画面を上下にスクロールさせた上で、別のページを開き、再度スクロールするという手順を繰り返した。その他のアプリでは、1 秒に 1 回程度手動で画面をスクロールさせた。以上のような操作を約 1 分間繰り返す試行を 10 回実施した。

4.1 描画処理数の推定が的中する確率

本研究の手法では、1 フレームに発生するアプリレベルの描画処理数を推定し、推定した数の描画処理が終了した場合に GPU を制御する。そのため、各フレームにおいて描画処理数の推定がどの程度正しかったかが重要である。

表 3 は GPU の状態制御の閾値を 0 に設定（残り時間が短くても状態が制御される閾値）した上で、実際のアプリを動作させた場合における描画処理数推定の的中率を示している。もっとも低いブラウザでも 80%程度の的中率となり、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十分認められる。

4.2 消費電力が削減可能なアイドル期間の長さ

本研究の提案方式では、1 フレーム分の描画処理が終了したと判定した場合に、その時点から 1 フレーム終了までの残り時間と閾値を比較して、GPU の状態を制御すべきか否かを決定しているが、フレームの残り時間が短いにもかかわらず GPU の状態を制御した場合、GPU の状態を変更したことにより削減される消費電力より、状態を変更

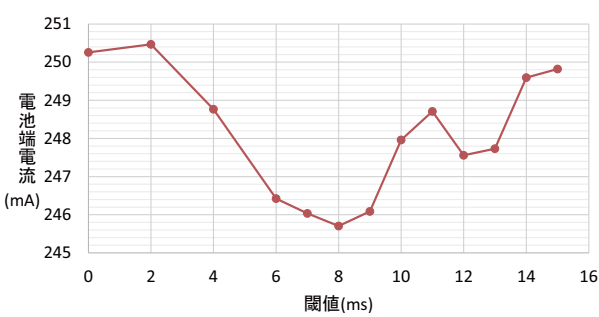


図 12 損益分岐点

したことに伴う、電力ロスが上回り、端末の電力消費が増加する可能性がある。しかし、閾値を大きくとると、電力ロスは減少するが、GPU の状態を制御できるフレームも減少する。実際には、閾値の値が「消費電力が削減可能なアイドル期間の長さ」と同じ値になっている場合が消費電力が最小となる。

「消費電力が削減可能なアイドル期間の長さ」を求めるため、実際のアプリを動作させ閾値を変化させながら繰り返し測定を行った。アプリケーションとしては、人の操作が介在せず周期的に長いアイドル期間を発生させるアプリとして youtube を用いた。図 12 は閾値を変化させて測定した結果である。この図からわかるように、1 フレーム（約 16.6ms）のうち残り時間が 8ms の時点を閾値に設定した場合に効果が最大となった。この 8ms という時間は、1 フレームの時間の約半分であるため、GPU の状態遷移による消費電力の削減効果が得られるアプリは、描画処理が早期に終了するものに限定される。

4.3 電力削減効果が見込めるアプリケーション

前節における評価から、Nexus5 では GPU の状態を制御して消費電力を削減するためには、1 フレームにおける描画処理終了から、次のフレーム開始まで 8ms 以上の時間が必要であることがわかったが、主要なアプリにおける 1 フレーム分の描画終了時刻を評価するため、GPU の状態変更の閾値と、GPU が低電力状態に留まる時間の関係を測定した。

この測定結果を集計したものが図 13 である。提案方式では、1 フレーム分の描画終了した時点から、次のフレームがはじまるまでの時間が閾値以上の場合に GPU を休止状態に変更する。この閾値（横軸）を変化させて、アプリケーション利用時間のうち、どの程度の時間 GPU が休止状態となるかを百分率で示したものである。例えば、閾値が 4 の場合は描画終了の時点で次フレーム開始までに 4ms 以上存在した場合に GPU を休止状態にする。その時、設定画面と youtube は、アプリ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

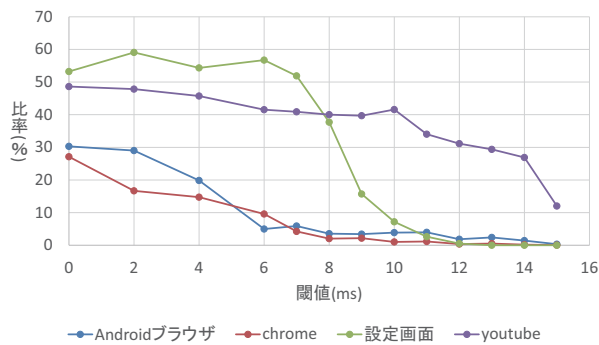


図 13 閾値と sleep 時間比率グラフ

表 4 各アプリにおける消費電流の削減効果		
アプリ名	電流削減値 (mA)	備考
youtube	4.4	手操作のため参考値
設定画面	15.8	
ホーム画面	7.6	

いる。

Nexus5 のハードウェアの場合、閾値は 8ms 以上にする必要があるが、このグラフからわかるように、2 種類のブラウザは次フレーム開始までの残り時間 8ms の時点までに描画を終了することはほとんどないため、閾値を 8ms に設定した場合は GPU を低電力状態に設定できる比率が非常に少ないことがわかる。それに対して、設定画面や youtube では、アプリ動作中の時間のうち、比較的多くの時間 GPU を低電力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。ブラウザで GPU の状態を制御して消費電力を削減するためには、閾値が 4ms もしくはそれ以下でなければならない。この時間は、GPU、電源回路やクロック発生回路などトータルのハードウェアで決まる時間であるため、値を半分にするためにはハードウェアも含めた見直しが必要である。

4.4 消費電流の削減効果

前節までの結果から、youtube および、端末設定画面では本研究の方式の効果が見込めるため、youtube と端末設定画面の他、端末設定画面とよく似た性質をもつホーム画面の合計 3 種類のアプリケーションについて、実際の端末に流れ込む電流を測定した（表 4）。

youtube 以外のアプリは人間による操作を行なっているため、データのばらつきがあるものの、低電力状態に長時間滞在する youtube よりも、消費電流削減効果が大きい。このような現象が発生する理由は、youtube 再生中の GPU の動作周波数は常に最低となるのに対して、設定画面やホーム画面のように間欠的に画面が変化するアプリでは、画面が変化した時点からしばらくの間、GPU の動作周波数が高くなるためであり、これは GPU のログおよびデバイスドライバの解読結果からわかっている。GPU の動作

周波数が増加した場合、待機状態と休止状態の場合の消費電流の差が大きくなり、その結果として、低電力状態に設定されている期間が短くても電流削減量が大きくなる。

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、1 フレームの描画処理終了後の GPU のアイドル期間の状態を制御するため、1 フレームの間に発生する、アプリレベル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 を低消費電力状態に設定する手法を提案し、提案方式のプロトタイプを Nexus5 で動作させて性能評価を行なった。

この結果から、ブラウザでは約 80%，その他のアプリでは 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することが判明した。

また、Nexus5 では GPU の電力を削減するためには、アイドル期間が少なくとも 8ms 以上必要であることがわかった。端末設定画面や 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、ブラウザではそれほど長時間のアイドル期間が発生せず、GPU の状態制御による電力削減は困難であることが判明した。

効果が見込めるアプリについて実際の端末の消費電流を測定したところ、youtube では平均 4.4mA、設定画面とホーム画面ではそれぞれ 15.8mA、7.6mA 端末に流れ込む電流を削減することができた。

参考文献

[1] Google: Android Developer, Google (online), available from <http://developer.android.com/> (accessed 2013-07-12).

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

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

[4] 長坂, 丸山, 額田, 遠藤, 松岡: FGPU 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, 情報処理学会, pp. 1-6 (2010).

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

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

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

[8] Jiao, Y., Lin, H., Balaji, P. and Feng, W.-c.: 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)*, IEEE, pp. 221–228 (2010).
- [9] 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*, IEEE, pp. 115–122 (2010).
 - [10] Suda, R. et al.: Power efficient large matrices multiplication by load scheduling on multi-core and GPU platform with CUDA, *Computational Science and Engineering, 2009. CSE'09. International Conference on*, Vol. 1, IEEE, pp. 424–429 (2009).
 - [11] 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, *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, IEEE, pp. 224–231 (2010).
 - [12] Google: Android NDK, Google (online), available from <https://developer.android.com/tools/sdk/ndk/index.html> (accessed 2014-02-24).
 - [13] Google: Nexus 5, Google (online), available from <http://www.google.co.jp/nexus/5/> (accessed 2014-02-24).