Vol.2012-OS-123 No.4
Vol.2012-EMB-27 No.4
2012/12/5

Android アプリケーションの起動時間に関する一考察

永田恭輔[†1]　　山口実靖[†1]

Android はスマートフォン，タブレット PC，音楽プレイヤーなど，様々なデバイスで採用されており，その重要性が年々増加している．Android において，アプリケーションの起動時間はユーザーにとって重要な性能の一つである．本稿ではまず，Android におけるアプリケーション起動手順と起動時間短縮のための仕組みを説明する．次に，我々が開発したアプリケーション起動時間の解説システムと起動時間の解析結果について述べる．そして，起動時間のさらなる短縮方法として事前読み込みクラス数の増加とアプリケーションプロセスの強制終了の回避を紹介し，その効果について考察する．

A Study on Application Launching Time in Android

KYOSUKE NAGATA[†1]　SANEYASU YAMAGUCHI[†1]

Android is applied on various devices such as smartphones, tablet PCs, and music players, and its users are increasing year by year. However, Android is still developing. Thus, its performance is not enough and has not been discussed in detail. Especially, application launching performance has not been analyzed enough even so it is one of the important performances for users. In this paper, we focus on application launch performance. Frist, we introduce our system which enables a profound investigation of an application launch procedure. Second, we show analyzing results of application launching with our system. After these, we introduce methods for improving application launching performance and discuss their affectivity.

1. はじめに

Android はスマートフォン，音楽プレイヤー，タブレット PC など様々なデバイスで動作する．Android の世界シェアは急速に増加しており，2012 年には 68 パーセントを超えた[1]．このように Android は非常に重要なプラットフォームとなっており，その性能向上は非常に重要であると言える．本稿では，ユーザーにとって特に重要な性能の一つであるアプリケーション起動性能に着目し，その短縮方法について議論する．

本論文構成は以下の通りである．2 章で Android とそのアプリケーション起動手順について紹介する．3 章では我々が提案した Android アプリケーション起動解析システムを紹介する．4 章ではアプリケーション起動時間の評価を行う．5 章ではアプリケーションの起動時間の短縮方法について考察する．6 章で関連研究の紹介を行い，結論を 7 章で述べる．

2. Android

2.1 Android アーキテクチャ

Android のアーキテクチャはカーネル，ライブラリ，Android ランタイム，アプリケーションフレームワーク，そしてアプリケーションの 5 つで構成されている[2]．Android ランタイムとアプリケーションフレームワークは Android のために新規に開発された構成要素を含んでおり，それらはまだ十分に調査されていない．また，これらの構成要素はアプリケーション起動に大きくかかわる．よって，アプリケーション起動時間の考察を行うにはこれらの動作の理解が重要であると言える．

カーネルはプロセス管理，メモリ管理，ネットワークスタックといったコア機能を提供し，これは Linux カーネルをベースに構築されている．したがって，その振る舞いは既存の Linux と類似していると期待される．

ライブラリは Android システムの様々なコンポーネントで利用されるソフトウェアである．これはアプリケーションフレームワークを通してアプリケーションによっても利用され，C 言語または C++言語で記述されている．ライブラリはシステム C ライブラリや SQLite といった基本的なコンポーネントを含んでおり，システム C ライブラリは BSD 由来の標準 C ライブラリ実装をもとにしている．これらのコンポーネントも，その他のオペレーティングシステム上と似た振る舞いをすると期待される．

Android ランタイムはコアライブラリや Dalvik VM から構成されている．これらは，Android 用に新規に実装され，アプリケーションの起動性能に大きな影響を与える．コアライブラリは Java プログラミングのための基本ライブラリである．Dalvik VM は Android アプリケーションを実行するための仮想計算機であり，アプリケーションの Dalvik バイトコードをインタープリトして実行する．

アプリケーションフレームワークはオープン開発プラットフォームを提供するためのコンポーネントを含んでおり，これらはデバイスアクセス，位置情報アクセス，バックグラウンドサービスの利用などの機能を提供している．Activity Manager のようなアプリケーション管理コンポーネントはこのフレームワークに含まれており，アプリケー

†1 工学院大学 大学院 工学研究科 電気電子工学専攻
Electrical Engineering and Electronics, Kogakuin University Graduate School

ションの起動に深い関係を持っている．またこれらは Android 用に新規に開発されものであり，まだ十分な考察が行われていないと考えられる．

2.2 Android アプリケーション

Android アプリケーションは，多くの場合 Java 言語で実装される．作成された Java ソースファイルは Java コンパイラによって Java VM 互換の.class ファイルへコンパイルされ，.class ファイルは Dalvik VM 互換の.dex ファイルへ変換される．.dex ファイルはリソースファイルと一緒にアプリケーションパッケージファイル（apk）へ圧縮される[3].

アプリケーション起動時に DavlikVM プロセスが.dex ファイルを読み込み，これを実行する．

2.3 Zygote

アプリケーション起動のオーバーヘッドを減少させるために，新しいアプリケーションプロセスは Zygote と呼ばれる特殊なプロセスからフォークされる．いくつかの重要なクラスファイルは，多くのアプリケーションから読み込まれるため，それぞれのアプリケーションが個別に読み込んでいたのでは効率が悪いと言える．これを改善するために Zygote プロセスは重要なクラスファイルを既にロードしており（preload しており），新しいアプリケーションプロセスはこの Zygote からフォークされることにより，クラスファイルをロードした状態で起動する．このように，Zygote はプロセス生成時間の短縮を実現している．

Android 4.1.1 で Zygote は 2281 個のクラスを読み込んだ状態で待機している。

2.4 アプリケーション起動手順

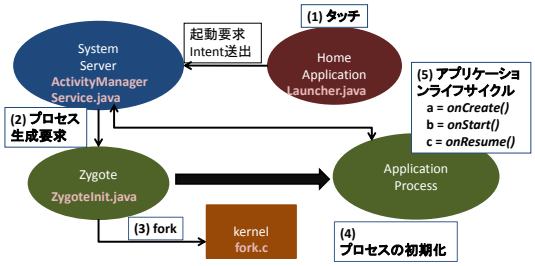

図 1. 新規起動におけるアプリケーションの起動手順

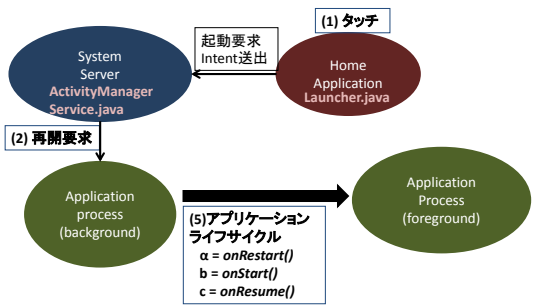

図 2. 再開におけるアプリケーションの起動手順

Android アプリケーションは以下の手順により起動される．アプリケーションの起動手順には 2 通りあり，どちらの手順にするかはプロセスの状態に応じて適用される．一つ目の手順は起動対象のプロセスが存在しない場合に，新しくプロセスを生成する手順である．2 つ目の手順は起動対象のアプリケーションプロセスがバックグラウンドプロセスとして既に存在している場合に，フォアグラウンドプロセスとして再開する手順である．本論文では，前者を「新規起動」，後者を「再開」と呼ぶ．

「新規起動」の場合，アプリケーションは以下の手順で起動される（図 1 参照）．(1)ユーザーはアプリケーションのアイコンをタッチする．その時，ホームアプリケーションが ActivityManager へ起動要求の Intent を送出する．(2)ActivityManager は Zygote へプロセス生成要求を送る．(3)Zygote は自身をフォークして新しいプロセスを生成する．(4)アプリケーションプロセスが初期化される．(5)アプリケーションライフサイクルの onCreate(), onStart(), onResume() が実行される．

「再開」の場合，アプリケーションは以下の手順により起動される（図 2 参照）．(1)ユーザーがアプリケーションのアイコンをタッチする．その時，ホームアプリケーションが ActivityManager へ起動要求の Intent を送出する．(2)ActivityManager はアプリケーションライフサイクルの onRestart(), onStart(), onResume()を呼ぶ．起動するアプリケーションプロセスはフォークや初期化が行われない．また，アプリケーションライフサイクルは onCreate()ではなく，onRestart()が実行される．

2.5 low memory killer

Android には low memory killer と呼ばれるアプリケーションプロセス強制終了プログラムが搭載されており，システムの空きメモリ量が規定量を下回ると同プログラムがアプリケーションプロセスを強制終了して使用可能メモリ量を増加させる．

low memory killer の動作を以下に示す．Android には *adj* と *minfree* の組が定義されている．Android 4.1.1 における *adj* と *minfree* の組の標準設定は表 1 の通りである．*adj* はアプリケーションのランクを表し，*adj* の値が高いアプリケーションから順に強制終了される．*adj* はアプリケーションの種類と状態により決定され，たとえば，フォアグラウンド状態にあるアプリケーションは *adj* が 0 であり，最

表 1 *adj* and *minfree* (Android 4.1.1)

adj	*minfree* [KB]
0	3674
1	4969
2	6264
4	8312
7	9607
15	11444

も強制終了の優先順位が低い．*minfree* はプロセス強制終了を行うか否かの空きメモリ量の閾値である．

たとえば空きメモリ量が 7000[KB]となった場合は，*adj*=2 の *minfree*(6264[KB])の条件を満たしているが，*adj*=4 の *minfree*(8312[KB])の条件は満たしていないため，空きメモリ量が 8312[KB]を上回るまで *adj* が 4 以上のプロセスを強制終了していく．その際，*adj* の値が高いアプリケーションを優先的に強制終了し，*adj* の値が同一のアプリケーションの中では消費メモリ量が大きいアプリケーションを優先的に強制終了していく．

起動済みのアプリケーションを再度起動する場合，アプリケーションが強制終了されていなければ前節の「再開」により起動されるが，強制終了された後であれば「新機起動」により起動される．多くの場合は前者の方が起動時間は短く、起動時間短縮のためには、強制終了の回避が重要と考えられる．

3. アプリケーション起動解析システム

この章では，我々が提案した Android アプリケーションの解析システム[4]について説明する．

まず，OS 内で発生したイベントをモニタリングする機能の実装を紹介する．このモニタリングシステムは処理のタイプ，イベントの時刻，プロセス ID，その他の処理の情報を記録する．カーネル部のモニタリング機能は Linux 用のカーネルモニタツールの実装を移植することにより構築した[5]．モニタリングシステムはイベントを記録するために前もってメモリを確保し，処理発生時に確保したメモリに処理の情報を保管する．モニタリングする過程はメモリにデータを保管するのみであり，モニタリングにおけるオーバーヘッドはとても小さい．ユーザー空間部の処理は，Android アプリケーションフレームワーク，Android ランタイム，ライブラリのソースコード内に，Android ログシステムのロギング関数を挿入してモニタした [6]．その保管した情報は Android Logcat コマンドによって見ることができる．モニタリング機能は Java 言語と C 言語で実装されている．前者は Java で記述されたアプリケーションフレームワークに挿入され，後者は C 言語で記述された Android ランタイムへ適用されている．

本モニタリングシステムにより，アプリケーション起動手順の全体を網羅的に見渡すことが可能になり，多くの時間を消費している部分を発見することが可能となる．

4. アプリケーション起動時間の評価

4.1 測定環境

表 2. 測定環境

Device name	HT03A(ADP2)	Nexus S
OS	Android2.1	Android4.0.1, Android4.1.1
CPU	QUALCOMM MSM7201a 528MHz	Cortex A8 (Hummingbird) processor 1GHz
Memory	192MB RAM	512MB RAM

この章では提案システムを用いた解析結果を示す．解析システムを構築した 2 つのスマートフォンの仕様は表 2 の通りである．これらの端末でアプリケーション起動をモニタする．

「新規起動」の場合，対象アプリケーションの既存プロセスを終了（kill）し，スクリーンをタッチしてアプリケーションを起動させる．「再開」の場合，事前に対象アプリケーションプロセスを起動しておき，ホームボタンを用いてバックグラウンドプロセスとしておく．そして，ホームボタンの後にスクリーンをタッチしてバックグラウンドプロセスをフォアグラウンドプロセスにしてアプリケーションを「再開」させる．

本論文では，アプリケーションの起動開始をスクリーンタッチと定義し，アプリケーションの起動完了をアプリケーションライフサイクルの onResume()が呼ばれた時と定義する．

4.2 総起動時間の測定

ブラウザアプリケーションの総起動時間を図 3 から図 6 に示す．横軸は何回目の起動かを示している．1 回目から 4 回目の起動が「新規起動」により起動されており，5 回目から 8 回目までの起動が「再開」により起動されている．

図 3，図 4 はオペレーティングシステムのキャッシュが有効の状態でアプリケーション起動したときの総起動時間である．これらより「新規起動」の総起動時間より「再開」総起動時間の方が短いことが分かる．また「新規起動」と「再開」のそれぞれの 1 回目の総起動時間より 2 回目以降の総起動時間の方が短いということも分かる．そして，Nexus S の総起動時間は HT-03A の約 6 分の 1 であることが分かる．

図 5，図 6 はオペレーティングシステムのキャッシュを無効にしたときの総起動時間を示している．図より，2 回目以降の総起動時間が 1 回目の総起動時間より短くないことが分かる．これらの結果より 2 回目以降の総起動時間が減少する直接的な原因がオペレーティングシステムのキャッシュであるということが分かる．

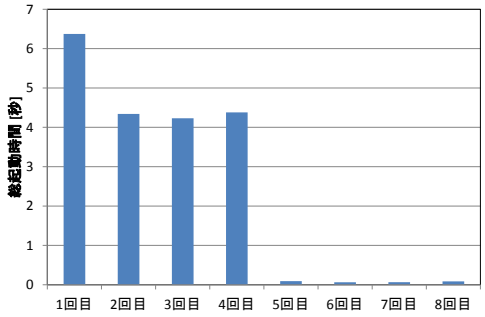

図 3. 総起動時間 (HT-03A, キャッシュ有効)

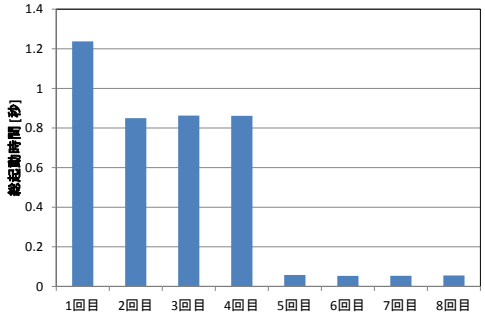

図 4. 総起動時間 (Nexus S, キャッシュ有効)

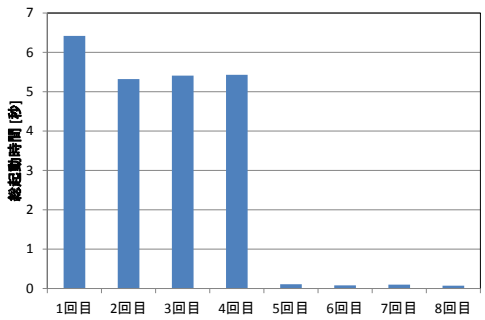

図 5. 総起動時間 (HT-03A, キャッシュ無効)

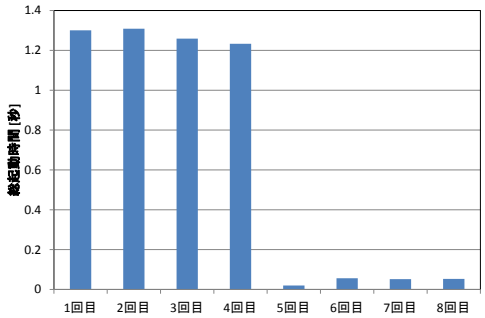

図 6. 総起動時間 (Nexus S, キャッシュ無効)

4.3 新規起動における起動時間の分割解析

本節では,「新規起動」でのアプリケーション起動にかかる総起動時間を各処理の時間に分割する.「新規起動」の手順は表 3 に示す通りである. そのため, 我々はこれと一致するように総起動時間を分割した. 手順内の各処理の所要時間を図 7 から図 10 に示す.

図 7 と図 8 より, (4)b から(5)a の初期化処理と, (5)a から(5)b の onCreate()の 2 つの処理が総起動時間の大半を占めていることが分かる. 加えて, この二つの処理の所要時間は 1 回目より 2 回目以降の方が短いことも確認できる.

次に, 分割された各処理におけるオペレーティングシステムのキャッシュについて考察する. 図 7, 図 8 はキャッシュ有効時の起動解析結果である. 図 9, 図 10 はキャッシュ無効時の解析結果である. キャッシュ有効時は, (4)b から(5)a の初期化処理と(5)a から(5)b の onCreate()処理の部分で 2 回目以降の時間短縮が確認された. 対照的に, キャッシュ無効時では時間の短縮は確認されなかった. よって, 初期化処理と onCreate()処理が最も時間を消費しており, オペレーティングシステムのキャッシュはこれらの処理で大きな効果があることが分かる.

表 3. 時刻を取得した各処理 (新規起動)

	時刻を取得した各処理	
(1)	スクリーンタッチ	
(2)	プロセス生成要求の送出	Intent が届くまでに要する時間
(3)	プロセス生成の開始	プロセス生成要求から生成開始までに要する時間
(4)a	アプリケーションの初期化処理 1	プロセス生成に要する時間
(4)b	アプリケーションの初期化処理 2	初期化処理に要する時間
(5)a	onCreate()が呼ばれる (アプリケーションライフサイクル)	初期化処理に要する時間
(5)b	onStart()が呼ばれる(アプリケーションライフサイクル)	onCreate()に要する時間
(5)c	onResume()が呼ばれる (アプリケーションライフサイクル)	onStart()に要する時間

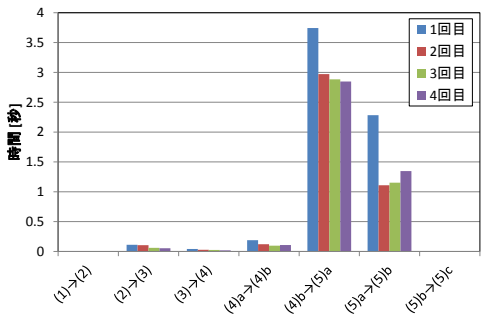

図 7. 分割した起動時間 (新規, HT-03A, キャッシュ有効)

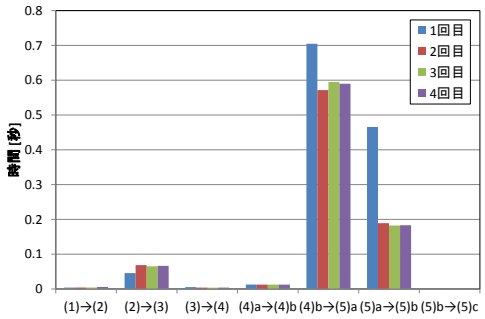

図 8. 分割した起動時間 (新規, Nexus S, キャッシュ有効)

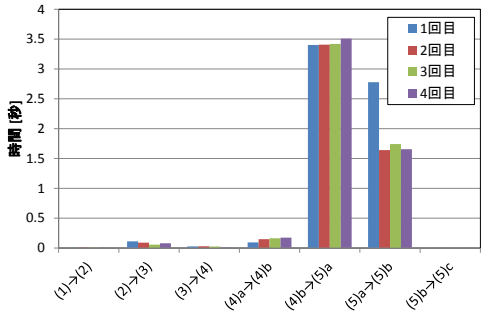

図 9. 分割した起動時間 (新規, HT-03A, キャッシュ無効)

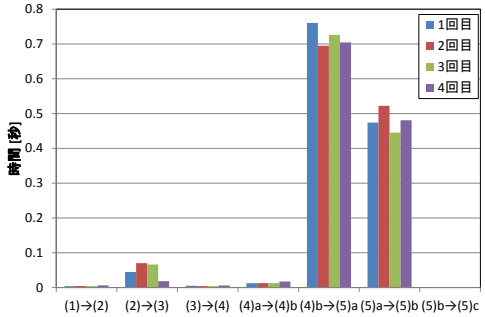

図 10. 分割した起動時間 (新規, Nexus S, キャッシュ無効)

4.4 再開における起動時間の分割解析

次に，「再開」における総起動時間の分割結果について述べる．モニタした各処理は表 4 のとおりである．結果を図 11 から図 14 に示す．これらの図から，(2)からαの処理に総起動時間のほとんどを費やしており，この部分においてキャッシュが効果的であることが確認できる．

図 7 と図 11 を比較すると，「新規起動」（図 7 参照）で時間のかかる 2 つの処理が「再開」の手順には無いことが分かりこれが「再開」の起動時間が「新規起動」の起動時間より非常に短くなる主な理由であることが分かる．また，「再開」手順より「新規起動」手順の方がよりキャッシュが効果的に働くということも確認された．

表 3. 時刻を取得した各処理 (再開)

	時刻を取得した各処理	
(1)	スクリーンタッチ	
(2)	再開要求	Intent が届くまでの時間
α	onRestart()が呼ばれる（アプリケーションライフサイクル）	再開要求から再開開始までに要する時間
(5)b	onStart()が呼ばれる(application lifecycle)	onRestart()に要する時間
(5)c	onResume()が呼ばれる（application lifecycle)	onStart()に要する時間

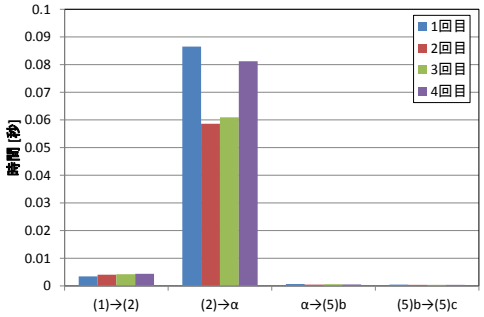

図 11. 分割した起動時間 (再開, HT-03A, キャッシュ有効)

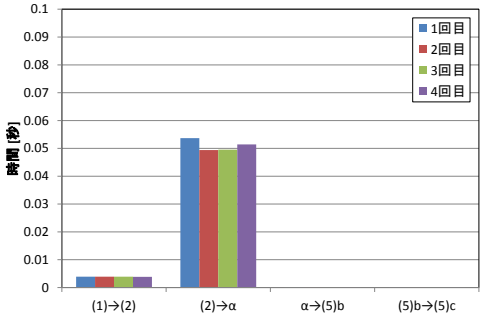

図 12. 分割した起動時間 (再開, Nexus S, キャッシュ有効)

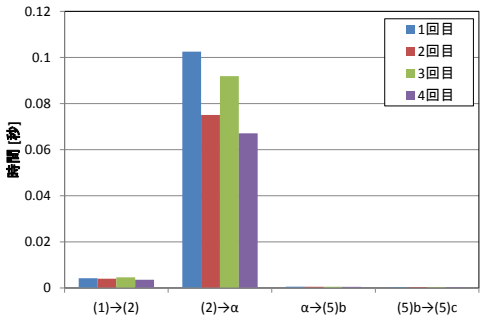

図 13. 分割した起動時間 (再開, HT-03A, キャッシュ無効)

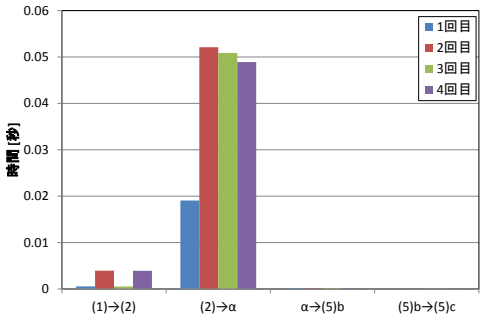

図 14. 分割した起動時間 (再開, Nexus S, キャッシュ無効)

5. アプリケーション起動時間の短縮

5.1 Zygote の preloading 機能の拡張

第 2.3 節で述べたように，Android OS には Zygote プロセスが存在し，このプロセスが多くのクラスファイルを読み

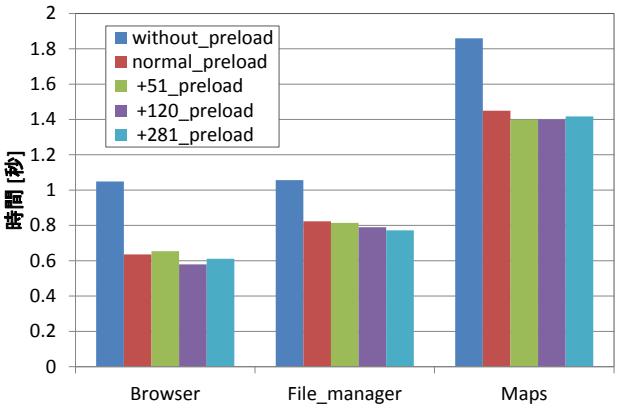

図 15. preload されるクラス数と起動時間の関係(A)

込み済みの状態で待機している．この Zygote プロセスが preload するクラスの数を増やすことにより，プロセス起動時間のさらなる短縮が実現できると予想される．本節では preload されるクラスの数とアプリケーション起動時間の関係について考察する．

図 15 に，Zygote が preload するクラスの量と起動時間の関係を示す．図内の“without_preload”は preload されるクラスを無くした場合，“normal_preload”は Android 4.1.1 の標準状態(2281 個)の場合，“+51_preload”は標準状態に加え dalvik 以下にあるすべてのクラス(51 個)を preload した場合，“+120_ preload”は 51_preload に加えさらに java.net 以下にあるすべてのクラス(合計 120 個)を preload した場合，“+281_preload”はさらに java.security 以下のすべてのクラス(合計 281 個)を preload した場合の測定である．測定は「新規起動」により行い，キャッシュが機能している状態でアプリケーションの起動を行った．

図より，Zygote が preload を全く行わないとアプリケーション起動時間が大幅に増加してしまうことが確認できる．よって，Zygote による preload はアプリケーション起動時間の短縮に効果があると言える．また，標準より多くのクラスを preload させることにより起動時間のさらなる短縮が可能であることがわかる．

次に，我々が個別に選択した 100 個のクラスと，我々が選択した 300 個のクラスを preload させた Zygote による実験も行った．結果を図 16 に示す．この図からも，Zygote に preload させるクラスの数の増加により，アプリケーションの起動時間のさらなる短縮が可能であることが分かる．

上記実験では，我々が重要と予想したクラス群を preload に追加して測定を行ったが，これらは必ずしも適切ではないと考えられる．アプリケーションが読み込むクラスの統計を取り，アプリケーションから実際に読み込まれることが多いクラス群を preload するよう変更すれば，さらに効率的にアプリケーション起動時間の短縮が可能になると予想される．

Zygote が preload するクラス数を増加させることによる使用メモリ量の増加について考える．すべてのアプリケーションプロセスは Zygote から fork されて作成されるが，Copy on Write を用いてプロセスがコピーされる．よって，書き込みが行われない限り，Zygote の preload 増加によるメモリ消費増加は他のアプリケーションプロセスの消費メモリの増加につながらない．また，これらクラスファイルに対しては書き込みが行われていない．よって，Zygote プロセスの消費メモリ増加分は他のプロセスの実消費メモリ量には影響を与えないと考えられる．

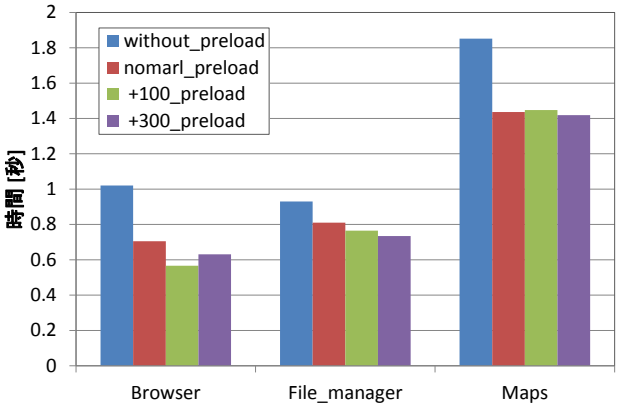

図 16. preload されるクラス数と起動時間の関係(B)

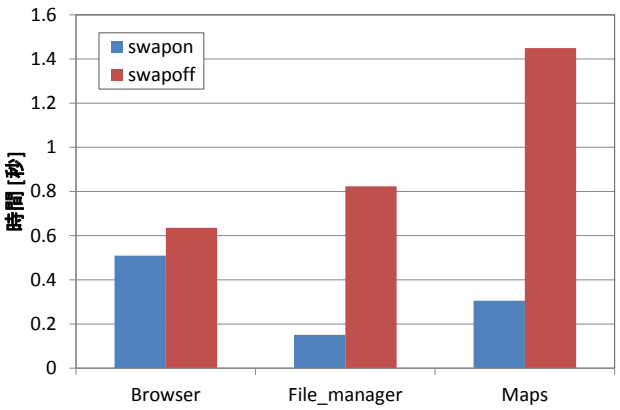

図 17. swapon と swapoff のアプリケーション起動時間

5.2 仮想メモリの使用によるプロセス終了の回避

前章で示した通り，「再開」によるアプリケーション起動の時間は,「新規起動」による起動時間より短い. よって，low memory killer によるアプリケーションの強制終了を回避できれば，起動済みアプリケーションを新規移動ではなく再開により起動することが可能となり，結果として起動済みアプリケーションの起動時間を短縮させられると考えられる．本節では，仮想メモリを用いて使用可能メモリ量を仮想的に増加させ，low memory killer によるプロセスの強制終了を抑制した場合におけるアプリケーション起動時間について考察する．

前節の Nexus S(実メモリ 512MB)において 1024MB の仮想メモリを確保し，*minfree* の値をすべて 1/100 倍にした環

境における起動済みプロセスの起動時間を調査した．仮想メモリを用いた場合と用いない場合のアプリケーション起動時間を図 17 に示す．ただし，実験は起動済みアプリケーションの起動後に 16 個のアプリケーションを起動してから行った．また，“swapoff”(仮想メモリ不使用)では low memory killer による強制終了の後に「新規起動」により起動しており，“swapon”(仮想メモリ使用)では「再開」により起動している．

図 17 より，“仮想メモリ使用”の方が起動時間が短く，low memory killer による強制終了の回避がアプリケーション起動時間の短縮に効果的であることが分かる．ただし，多くのアプリケーションをバックグラウンドプロセスとして残したままにしておくと，それらプロセスによりフォアグラウンドプロセスの実行性能が低下するとも考えられる．よって，仮想メモリの使用と low memory killer の閾値の減少は，アプリケーション起動性能とフォアグラウンドプロセス性能の両方を考慮して行うべきであると考えられえる．

6. 関連研究

カーネルをモニタリングするための既存研究として，カーネルモニタリングツールの FTrace[7][8]，SystemTap[9][10]LTTng[11][12]，OProfile[13]がある．これらのツールを用いることにより，Linux カーネルの内部処理の観察が可能になる．しかし，それらは Linux 用に構築されているため Android の解析には適さず，アプリケーションフレームワークや Dalvik VM の解析ができない．また，我々の提案手法は，オーバーヘッドが非常に小さいが，既存のこれらの手法は処理への影響が小さくない．

ユーザー空間部のプロファイラとして，Valgrind のようないくつかの研究がある[14]．それらはアプリケーション性能の深い解析が可能であるが，解析がカーネル部には及んでいない．

Android 性能に関するものとして以下の研究がある．[15]では Android の仮想計算機と通常の Java の仮想計算機の性能が比較されており，電力消費について議論されている．[16]では C 言語で記述されたネイティブアプリケーションの性能と DalvikVM 上で動いているアプリケーションの性能が議論されている．そして，性能ではネイティブアプリケーションが非常に有利であることを示している．しかし，これらの研究は Android アプリケーションの起動性能には着目していない．そのため，それらの研究成果は主旨が我々のものと異なる．

サーバーコンピュータやネットワークストレージなどを含む網羅的な解析手法は[5]で提案されている．この手法を用いて iSCSI のストレージアクセスの End-to-End 性能を解析することが可能である．しかし，この手法は iSCSI のストレージアクセスに限ったものであり，Android アプリケーションの起動の解析には適用できない．

7. おわりに

本論文で我々は Android アプリケーションの起動時間の解析システムを紹介し，アプリケーションの起動時間の解析結果を示した．そして，アプリケーション起動時間のさらなる短縮方法として，Zygote による読み込みクラス数を増加させる手法と，仮想メモリを有効化して low memory killer によるアプリケーション強制終了の閾値を減少させる手法を紹介した．

実験の結果，これらの手法に効果があることが確認された．今後は，Zygote が preload するクラスの選定に関する考察，バックグラウンドプロセスによるフォアグラウンドプロセスへの影響について考察していく予定である．

謝辞

本研究は科研費(22700039, 24300034)の助成を受けたものである．

参考文献

1) Android and iOS Surge to New Smartphone OS Record in Second Quarter, According to IDC : http://www.idc.com/getdoc.jsp?containerId=prUS23638712
2) What is Android?|Android Developers: http://developer.android.com/guide/basics/what-is-android.html
3) Dalvik (software) http://en.wikipedia.org/wiki/Dalvik_%28software%29
4) 永田 恭輔，山口 実靖，“Android アプリケーションの起動性能解析システムとその評価”，マルチメディア、分散、協調とモバイル DICOMO2012 シンポジウム，pp. 83 - 90, 2012
5) Saneyasu Yamaguchi, Masato Oguchi, Masaru Kitsuregawa, "Trace System of iSCSI Storage Access," IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2005), pp. 392-398, (2005)
6) logcat|Android Developers http://developer.android.com/guide/developing/tools/logcat.html
7) Steve Rostedt. ftrace tracing inftrastructure. http://lwn.net/Articles/270971/. SystemTap http://sourceware.org/systemtap/
8) Frank Ch. Eigler. “Problem solving with systemtap,” In Proceedings of the Ottawa Linux Symposium 2006, 2006.
9) LTTng Project http://lttng.org/
10) T. Bird, “Measuring Function Duration with Ftrace,” in Proc. of the Japan Linux Symposium, 2009.
11) M. Desnoyers, M. R. Dagenais, "The LTTng Tracer: A low impact performance and behavior monitor of GNU/Linux" Proceedings of Ottawa Linux Symposium 2006, 2006, pp. 209-223.
12) J. Levon and P. Elie. Oprofile: A system profiler for linux. http://oprofile.sf.net, September 2004.
13) Valgrind http://valgrind.org
14) Kolin Paul, Tapas Kumar Kundu “Android on Mobile Devices: An Energy Perspective,” 10th IEEE International Conference on Computer and Information Technology, 2010.
15) Leonid Batyuk, Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Ahmet Camtepe and Sahin Albayrak, “Developing and Benchmarking Native Linux Applications on Android,” Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 2009, Volume 7, 381-392