情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2013-GN-86 No.22
Vol.2013-CDS-6 No.22
2013/1/17

# クラス先読みの拡張による Android アプリケーション起動時間の改善

永田恭輔[†1]　服部拓也[†1]　中村優太[†2]　野村駿[†2]　山口実靖[†2]

Android スマートフォンにおいて，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となっている．しかし，Android アプリケーションは特殊な起動手順により起動されるため，その動作の詳細な解析が実現されておらず，起動時間の短縮に関する研究はほとんどなされていない．本研究では，Android 端末におけるアプリケーション起動時間の短縮を目的として，まずアプリケーション起動時間の解析システムの提案と構築を行う．具体的にはアプリケーション起動手順にて使用される Android 内の実装(Android カーネル，アプリケーションフレームワークなど)にイベント発生時刻を記録する機能を追加し，起動時間の調査を実現する．そして，Zygote の読込済み(preload)クラスの拡大，アプリケーション固有クラス(非標準クラス)の preload によりアプリケーション起動時間を短縮する手法を提案する．アプリケーション起動時間の解析システムを実装して評価したところ，提案解析システムによりアプリケーション起動時間の詳細解析が可能となり，多くの時間を消費している箇所の特定が可能であることが分かった．また，起動時間短縮手法を実装して評価したところ，preload クラスの拡大手法とアプリケーション固有クラスの preload の手法にて起動時間の短縮が見られ，提案手法の有効性が確認された．

# A Study on Application Launching Time in Android

KYOSUKE NAGATA[†1]　TAKUYA HATTORI[†1]　YUTA NAKAMURA[†2]
SHUN NOMURA[†2]　SANEYASU YAMAGUCHI[†2]

Application launching time is one of the most important performances for Android smartphone users. However, launching performance is not discussed enough because Android applications are launched based of its specialized launching procedure. In this paper, we discuss Android application launching time. First, we have proposed an application launching time analyzing system and have shown analytical results of practical applications. Second, we have proposed application launching time improvement methods, which have enhances preload function of Zygote process. Our experiments have demonstrated that the proposed methods have been able to improve application launching time.

## 1. はじめに

スマートフォン，タブレット型端末が急速に普及し，これらのデバイスのプラットフォームである Android に注目が集まっている．Android は登場以来の世界シェアを増加させ続けており，2012 年には 68 パーセントを超えている[1]．Android は非常に重要なプラットフォームとなっており, その性能向上は重要な課題であると言える. 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，その解析方法と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 Android アプリケーション起動解析システムを提案する．4 章では，提案解析システムを用いて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を提案し，評価する．6 章で関連研究の紹介を行い，結論を 7 章で述べる．

†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
†2 工学院大学　工学部　情報通信工学科
Department of information and Communications Engineering, Kogakuin University

## 2. Android

### 2.1 構成

Android 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研究，考察されていない状況にあると言える．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，これまでの研究成果を効果的に利用できると予想される．

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．これら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

ⓒ 2013 Information Processing Society of Japan

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．よって，これらの要素の詳細な解析の実現がアプリケーション起動時刻の考察や短縮には重要であると考えられる．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．よって，これらの詳細な解析が起動時間の解析には必要と考えられる．

### 2.2 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 fork 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みを行っていると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．よって，アプリケーションプロセスはクラスファイルをロード済みの状態で起動される．これにより，アプリケーション起動時間の短縮が実現されている．

また，アプリケーションの fork は CoW (Copy on Write) で行われるため，プロセスが生成された時点では親子のプロセスでメモリを共有しており，消費メモリ量が増加しない．通常は読み込まれたクラスファイルへの書き込みは行われないため，CoW でコピーされた領域はプロセスの終了までコピーされることはなく，メモリ使用量の削減も実現されている．

Zygote が preload するクラスファイルはフレームワーク内の `preloaded-classes` にて規定されており，Android 4.1.1 では 2303 個のクラスが Zygote により preload されている．

### 2.3 アプリケーション起動手順

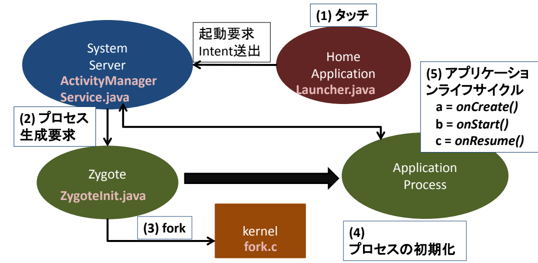


図 1. 新規起動におけるアプリケーションの起動手順

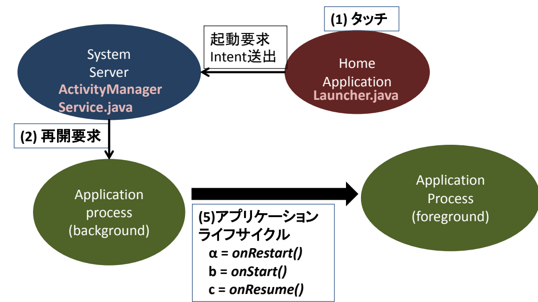


図 2. 再開におけるアプリケーションの起動手順

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，どちらの手順が適用されるかは，プロセスの状態により定まる．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)ユーザーがアプリケーションのアイコンをタッチする．そして，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 が Zygote へプロセス生成要求を送る．(3)Zygote が自身を fork 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．

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

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

「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．そして，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 が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では `onCreate()`ではなく，`onRestart()`が実行される．

**2.4 low memory killer（LMK）**

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[page]となった場合は，*adj*=2 の *minfree*(6264[page])の条件を満たしているが，*adj*=4 の *minfree*(8312[page])の条件は満たしていないため，空きメモリ量が 8312[page]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いため，起動時間短縮のためには強制終了の回避が重要である．

## 3. アプリケーション起動解析システム

本章にて，Android アプリケーション起動プロセスの解析システムを提案する．

Android アプリケーションは，前章に示した手順により起動される．よって，前章で示した各イベントが発生した時刻を調査することにより，アプリケーションの起動時間や，起動処理の中で特に多くの時間を消費している箇所を特定できる．本提案手法では，オープンソースである Android システムの該当実装部に調査機能を追加することにより実現した．実装の詳細を以下に示す．

まず，OS 内で発生したイベントをモニタリングする．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 Android に移植することにより構築した[4][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．モニタリングする処理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 コマンドによって取得することができる. モニタリング機能は Java 言語と C 言語で実装されており，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，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」の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．

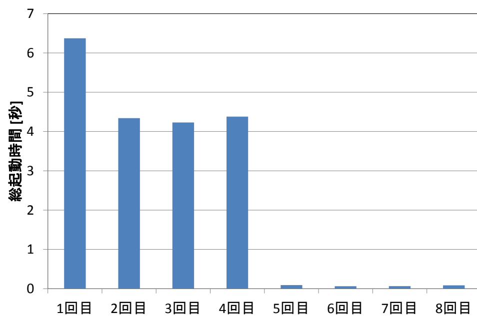


図 3. 総起動時間（HT-03A，キャッシュ有効）

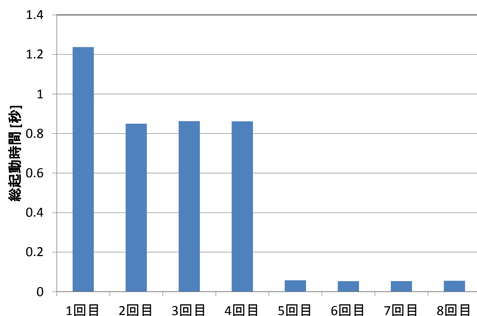


図 4. 総起動時間（Nexus S，キャッシュ有効）

### 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して示す．「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

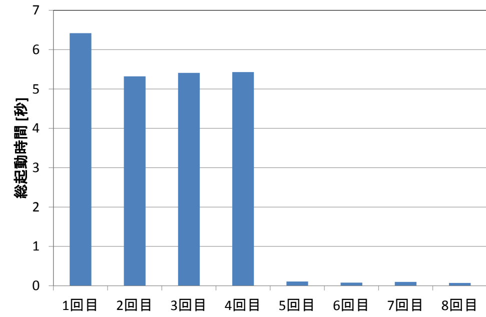


図 5. 総起動時間（HT-03A，キャッシュ無効）

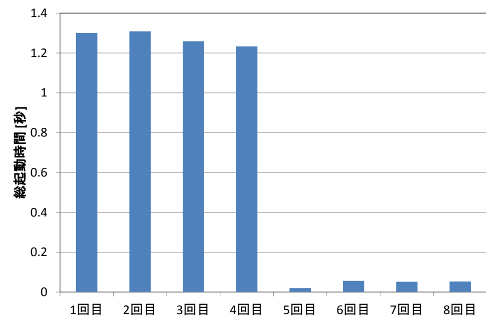


図 6. 総起動時間（Nexus S，キャッシュ無効）

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対するオペレーティングシステムのキャッシュの影響について考察する．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に対して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 | |
| | | Intent が届くまでに要する時間 |
| (2) | プロセス生成要求の送出 | |
| | | プロセス生成要求から生成開始までに要する時間 |
| (3) | プロセス生成の開始 | |
| | | プロセス生成に要する時間 |
| (4)a | アプリケーションの初期化処理 1 | |
| | | 初期化処理に要する時間 |
| (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） | |
| | | onCreate()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) | |
| | | onStart()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） | |

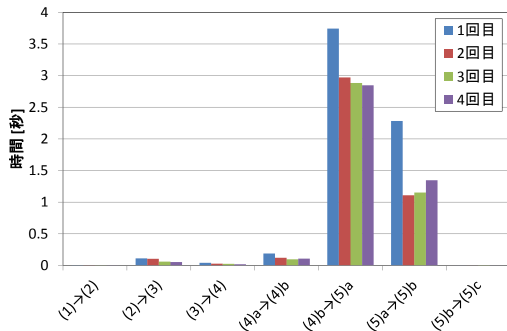


図 7. 分割した起動時間 (新規, HT-03A, キャッシュ有効)

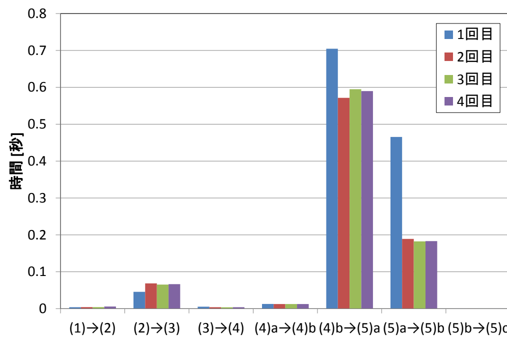


図 8. 分割した起動時間 (新規, Nexus S, キャッシュ有効)

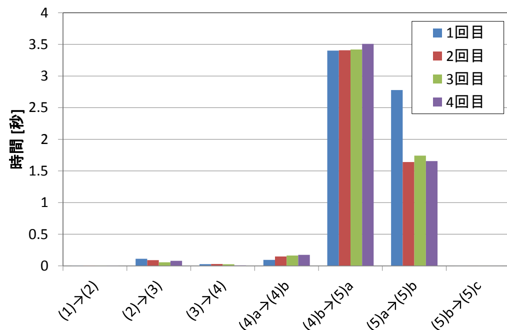


図 9. 分割した起動時間 (新規, HT-03A, キャッシュ無効)

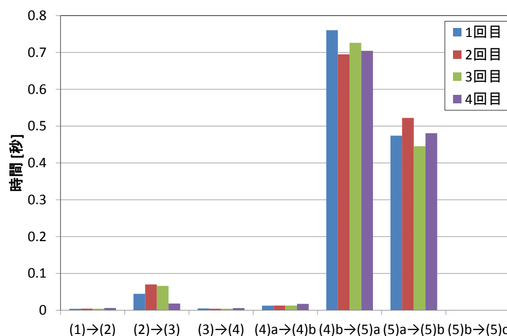


図 10. 分割した起動時間 (新規, Nexus S, キャッシュ無効)

### 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が費やされており，この部分においてキャッシュが効果的に機能することが確認できる．

図 7 と図 11 を比較すると，「新規起動」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 | Intent が届くまでの時間 |
| (2) | 再開要求 | 再開要求から再開開始までに要する時間 |
| α | onRestart()が呼ばれる (アプリケーションライフサイクル) | onRestart()に要する時間 |
| (5)b | onStart()が呼ばれる(application lifecycle) | onStart()に要する時間 |
| (5)c | onResume()が呼ばれる (application lifecycle) | |

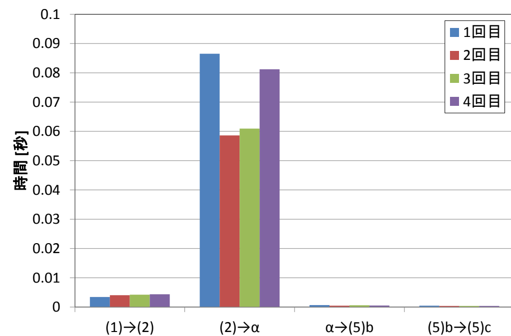


図 11. 分割した起動時間 (再開, HT-03A, キャッシュ有効)

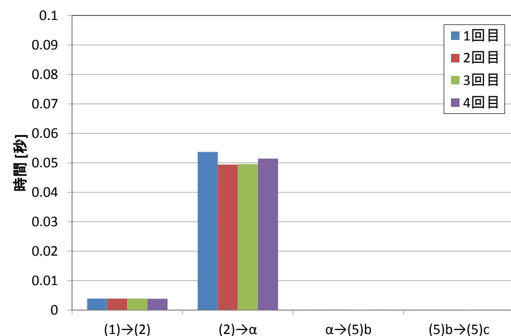


図 12. 分割した起動時間 (再開, Nexus S, キャッシュ有効)

ⓒ 2013 Information Processing Society of Japan

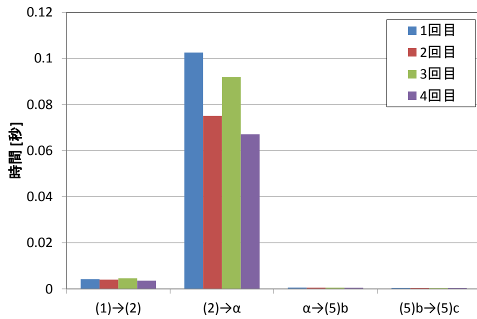


図 13. 分割した起動時間（再開, HT-03A，キャッシュ無効)

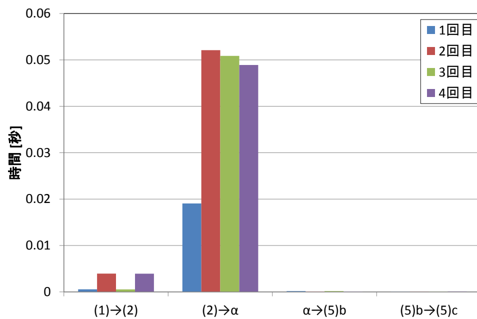


図 14. 分割した起動時間（再開, Nexus S，キャッシュ無効)

## 5. アプリケーション起動時間の短縮

### 5.1 クラス先読み

#### 5.1.1 標準クラスの先読み機能の拡張

第 2.2 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考えられる．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(2303 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 による実験を行った．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．

#### 5.1.2 アプリケーション固有クラスの先読み

次に，アプリケーション固有のクラスを Android の framework に組み込み，Zygote の preload クラスに追加することによる起動時間を短縮させる手法について考察する．標準ブラウザと Settings の 2 種類のアプリケーションについて，それらのアプリケーションの固有クラスを Zygote に preload させ起動時間を評価した．標準ブラウザに関してはキャッシュ有効時とキャッシュ無効時で比較を行い，Setting に関してはキャッシュ有効の状態で評価を行った．結果を図 17 に示す．図より，アプリケーション固有のクラスを Zygote に組み込む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確認され，当該クラスがキャッシュ内に格納されていないときに特に効果が大きいことが確認された．

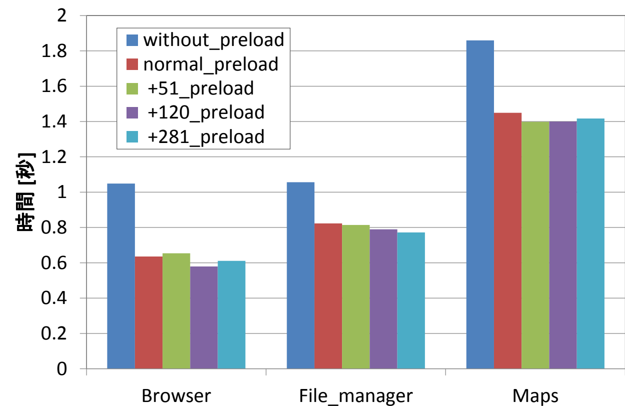


図 15. preload されるクラス数と起動時間の関係(A)

### 5.2 Low Memory Killer（LMK）の抑制

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，また low memory killer によるプロセスの強制終了の閾値を大幅に減少させて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．

ⓒ 2013 Information Processing Society of Japan

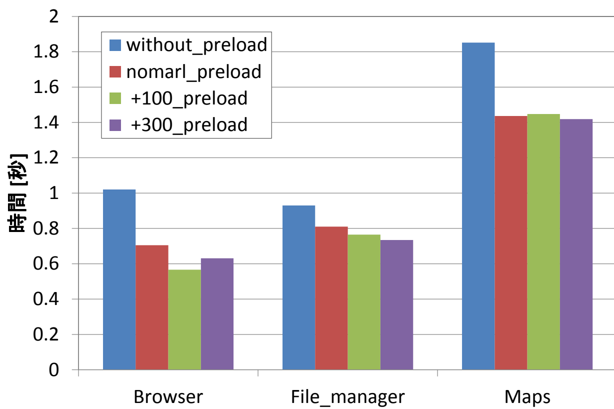


図 16. preload されるクラス数と起動時間の関係(B)

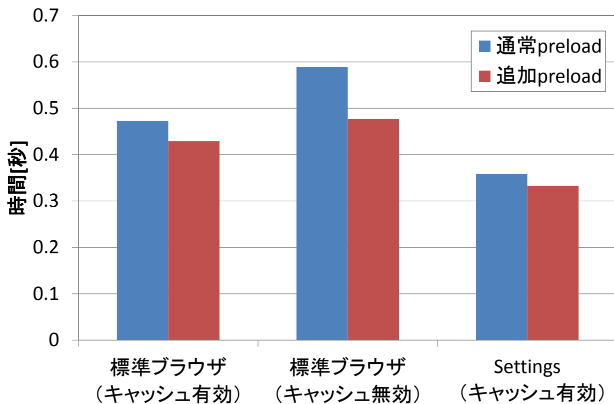


図 17. アプリケーション固有クラスの preload

前節の Nexus S(実メモリ 512MB)において 1024MB の仮想メモリを作成し，*minfree* の値をすべて 1/1000 倍にした環境におけるアプリケーション起動時間を調査した．以下において，“minfree*1”は通常の *minfree* の状態を，“minfree*1/1000”は *minfree* を 1/1000 倍にした状態を，“swap0GB”は仮想メモリを使用していない状況を，“swap1GB”は 1GB の仮想メモリを使用としている状態を意味している．

図 18 は累積起動プロセス数と残存プロセス数の関係を示している．起動したのは自作のベンチマーク用アプリケーションであり，onCreate にて 30MB のメモリを確保し待機するものである．仮想メモリ不使用かつ通常の *minfree* の状態では，5 個目のアプリケーションが起動された時点で 1 個目に起動されたアプリケーションが low memory killer により強制終了されるため，残存プロセス数は 4 個で頭打ちとなっている．一方，仮想メモリを使用し *minfree* を 1/1000 倍にしたときは残存プロセスは 14 個まで延びることが確認できる．このときの仮想メモリ使用量は図 19 の通りである．これらのことより，仮想メモリを使用し low memory killer の *minfree* の値を小さくすることにより，low memory killer によるアプリケーションの強制終了を回避することであることが分かる．

図 20，図 21 はアプリケーション起動時間の測定結果であり，図 20 が標準ブラウザの起動時間，図 21 が上記の自作アプリケーションの起動時間である．図 22 では新規起動，再開ともに LMK 抑制時の起動時間が長くなっているが，LMK の抑制によりプロセス強制終了を回避することが可能であり，通常時（minfree*1）には新規起動となる状態でも抑制時（minfree*1/1000）には再開とすることができる．これは，通常時における新規起動（約 0.25 秒）を抑制時の再開（約 0.15 秒）に変更できたことを意味し，大幅な短縮が実現されていると言える．

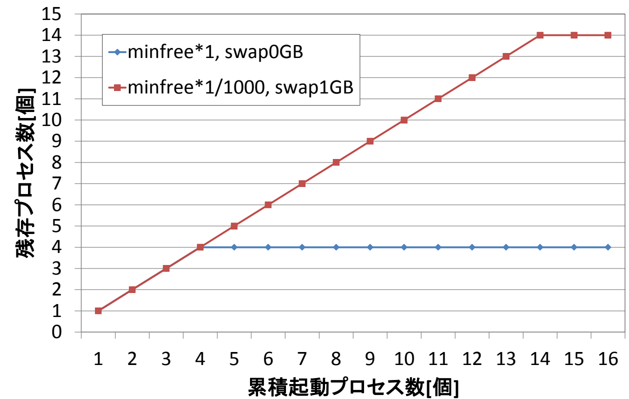


図 18. 起動プロセス数と残存プロセス数の関係

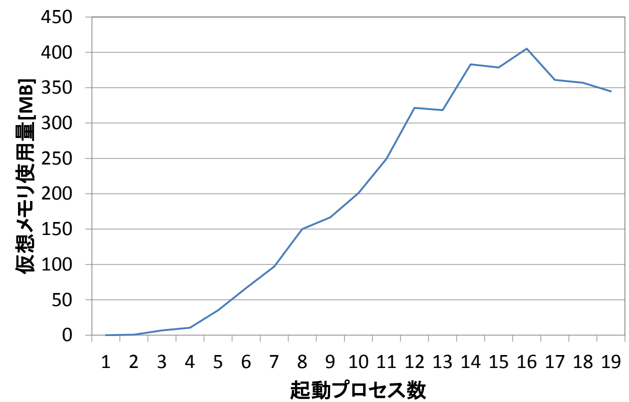


図 19. 起動プロセス数と仮想メモリ使用量

一方図 21 においては，LMK 抑制時の再開の時間が通常時の新規起動の時間より長くなっており，抑制による効果が表れていない．これはバックグラウンドプロセスを多く保有することによる負荷の増大が原因と考えられる．

以上より，low memory killer による強制終了の回避によりアプリケーション起動時間の短縮が可能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ことも考えられる．また，この性能低下により再開に要する時間が新規起動に要する時間を上回ってしまうこともある．

よって，仮想メモリの使用と low memory killer の閾値の

ⓒ 2013 Information Processing Society of Japan

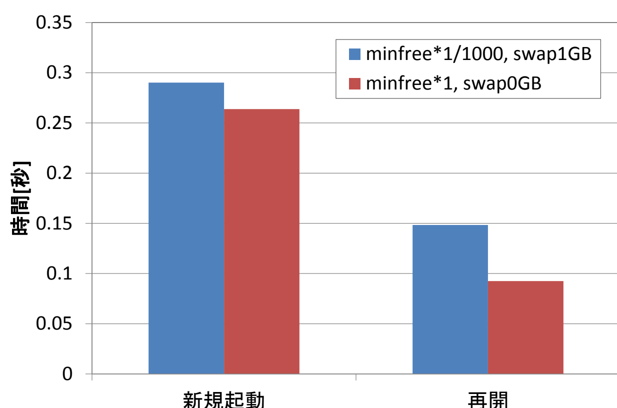


図 20. minfree の変更によるアプリケーション起動時間(A)

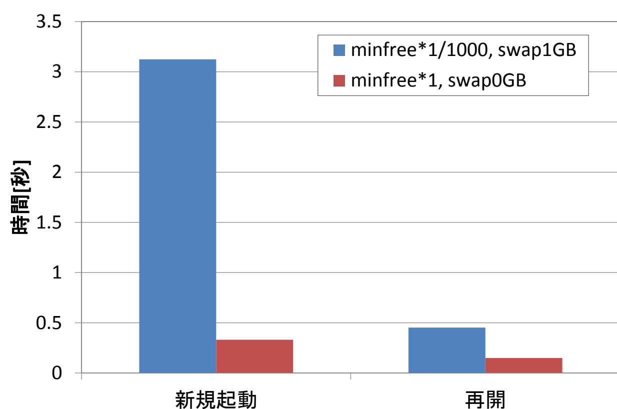


図 21. minfree の変更によるアプリケーション起動時間(B)

減少は，アプリケーション起動性能やフォアグラウンドプロセス性能なども考慮して行うべきであると考えられえる．

## 6. 関連研究

Android 性能に関するものとして以下の研究がある．文献[7]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．文献[8]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．また，[9]では消費電力を考慮しての性能制御手法が提案されている．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．

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[10][11]，SystemTap[12][13]LTTng[14][15]，OProfile[16]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．

## 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提案し，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．そして実験により，これらの手法に効果があることを示した．

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) Kolin Paul, Tapas Kumar Kundu “Android on Mobile Devices: An Energy Perspective,” 10th IEEE International Conference on Computer and Information Technology, 2010.
8) 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
9) Kyosuke Nagata, Saneyasu Yamaguchi, "An Android Application Launch Analyzing System ," 8th ICCM: 2012 International Conference on Computing Technology and Information Management, 2012
10) Steve Rostedt. ftrace tracing inftrastructure. http://lwn.net/Articles/270971/. SystemTap http://sourceware.org/systemtap/
11) Frank Ch. Eigler. “Problem solving with systemtap,” In Proceedings of the Ottawa Linux Symposium 2006, 2006.
12) LTTng Project http://lttng.org/
13) T. Bird, “Measuring Function Duration with Ftrace,” in Proc. of the Japan Linux Symposium, 2009.
14) 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.
15) J. Levon and P. Elie. Oprofile: A system profiler for linux. http://oprofile.sf.net, September 2004.
16) Valgrind http://valgrind.org

ⓒ 2013 Information Processing Society of Japan