Vol.2016-GN-97 No.28
Vol.2016-CDS-15 No.28
Vol.2016-DCC-12 No.28
2016/1/22

Android アプリケーションの状態応じた GC アルゴリズム切り替えに関する一考察

濱中真太郎[†1] 坂本寛和[†1] 小口正人[†2] 山口実靖[†1]

Lollipop 以降の Android OS では，アプリケーション実行環境として仮想マシン Android Runtime(ART)が採用されている．ART のメモリ管理機能のひとつに GC(Garbage Collection)機能があり，この動作がアプリケーションの性能に影響を与える．ART はアプリケーション実行時に GC のアルゴリズムを切り替える機能を有しており，状態に応じて適切な GC 実装を選択することが可能である．また，Android には OS 内の空きメモリ容量が閾値より少なくなるとアプリケーションプロセスを強制終了する Low Memory Killer という機能があり，アプリケーションプロセスの消費メモリの増加はアプリケーションプロセスの強制終了回数の増加につながる．本稿では，ART の GC 実装である CMS GC と SS GC に着目し，これらの GC の基本性能評価結果を示す．そして，アプリケーションの消費メモリ量やアプリケーションプロセスの状態(フォアグラウンド/バックグラウンド)に応じて GC を選択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．

1. はじめに

Lollipop 以降の Android OS では，アプリケーション実行環境として仮想マシン Android Runtime(ART)が採用されている．Android 上で動作するアプリケーションは原則的に ART 上で動作し，メモリ管理等は ART が行う．ART のメモリ管理機能のひとつに GC(Garbage Collection)機能がある．GC は参照されてないオブジェクトを見つけ開放を行う機能である．GC の処理はアプリケーション性能に影響を与える[1]．本稿ではバージョン 5.0 以降の Android に実装されている GC アルゴリズムの中で CMS(Concurrent Mark Sweep)GC と SS(Semi Space)GC に着目し，それらの振る舞いや性能についての考察を行う．

また，Android には，OS 内の空きメモリ容量が閾値より少なくなるとアプリケーションプロセスを強制終了する Low Memory Killer という機能がある．強制終了されたアプリケーションを再度起動するには，プロセスの再生成が必要となり長い時間を要する[2]．

本稿ではまず，ART の GC 実装のである CMS GC と SS GC に着目し，これらの GC の基本性能評価結果を示す．そして，適した GC はアプリケーションプロセスの消費メモリサイズやアプリケーションプロセスの状態(フォアグラウンド/バックグラウンド)により異なることを示す．次に，上記考察を踏まえアプリケーションの消費メモリ量やアプリケーションプロセスの状態に応じて GC を選択する手法を提案する．

2. Android

2.1 ART(Android Runtime)の GC

ART には CMS，SS，Generational Semi Space などの複数の GC アルゴリズムが実装されており，開発者が選択することができる．ただし，動作が安定しない実装も多く，本稿では安定的な動作が確認された CMS GC と SS GC に着目して考察を行う．標準で採用されているアルゴリズムは CMS GC である．

CMS GC と SS GC の処理とアプリケーションの停止タイミングを図 1 に示す. GC の各フェイズの呼称は Android のソースコードに記述されているものを採用している．

CMS GC の動作は次の通りである．まず，CMS GC はルートオブジェクトが参照しているオブジェクトにマークを付ける．次にそのオブジェクトが参照しているオブジェクトにマークを付ける．以下同様に再帰的にオブジェクトのマークを付ける．マークが付けられなかったオブジェクトはゴミオブジェクトとして回収され，それらの領域を再度利用可能な状態になる．

SS GC の動作は次の通りである．CMS と同様に参照されているオブジェクト全てにマークをつける．そしてマークされたオブジェクトを別領域にコピーをしヒープ領域の移行を行う．移行に伴い，旧領域は破棄される．よってこの別領域へのコピー処理にてコピーされなかったオブジェクトは領域ごと破棄される．

ART はアプリケーション実行時に GC のアルゴリズムを切り替える機能を有している．この機能は標準では有効になっておらず，ビルドオプションを変更して OS を再構築することにより有効にすることができる．初期設定である本ビルドオプション無効時は，アプリケーションフォアグラウンド時，バックグラウンド時ともに CMS GC が使用さ

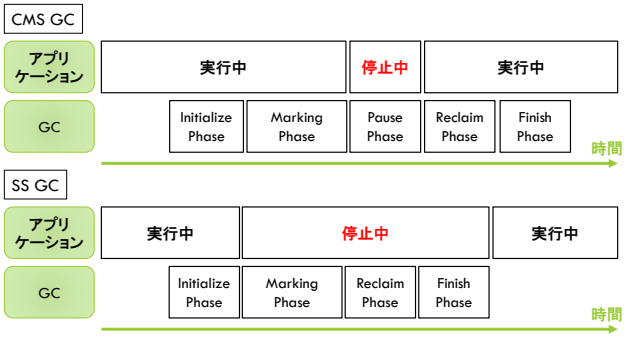

図 1. GC フェイズとアプリケーション停止時間

†1 工学院大学大学院 電気電子工学専攻
†2 お茶の水女子大学

Vol.2016-GN-97 No.28
Vol.2016-CDS-15 No.28
Vol.2016-DCC-12 No.28
2016/1/22

れ，ビルドオプション有効時の初期設定ではフォアグラウンド状態で CMS GC，バックグラウンド状態で SS GC が使用される．

2.2 Low Memory Killer

Android には，low memory killer という独自のプロセスメモリ管理システムが搭載されている[2]．これは，システム全体のメモリ空き容量が閾値以下まで下がった場合に起動され，プロセスを強制終了する機能である．多くの携帯端末同様に，Android はユーザがアプリケーションの終了処理を行わずに使用できるように設計されている．すなわち，ユーザはそのときに使用したいアプリケーションの起動(開始)のみを行えば良く，アプリケーションの終了操作を行う必要がない．よって，ユーザがアプリケーションを終了させることなく稼働アプリケーション数を増やし続けると OS が管理する空きメモリ量が単調に減り続け，必然的にシステムはメモリ不足の状態に陥る．Low Memory Killer はこの問題を解決するために(換言すると，ユーザにアプリケーション終了操作を要求しないシステムを実現するために)用意されており，システム全体の空きメモリ量が少なくなると自動的にアプリケーションを終了する．

2.3 アプリケーションのライフサイクル

Android アプリケーションのライフサイクルを図 2 に示す．中段の 3 個の楕円はアプリケーションプロセス状態を示しており，それぞれアプリケーションプロセスが無い状態，フォアグラウンドにアプリケーションプロセスがある状態，バックグラウンドにアプリケーションプロセスがある状態を示している．プロセス無しの状態からアプリケーション起動要求が発生した場合プロセスの新規作成が行われ `onCreate()` メソッドが呼ばれる．そしてアプリケーションのプロセスがフォアグラウンドになる．アプリケーションプロセスがフォアグラウンドの状態で別のアプリケーションが起動されると元々フォアグラウンド状態であったアプリケーションは画面上に表示されないバックグラウンド状態に移行する．アプリケーションプロセスがバックグラウンドにあるときにそのアプリケーションの起動要求が発生すると `onResume()` メソッドが呼ばれアプリケーションプロセスの再開処理が行われる．既存のアプリケーションプロセスを用いてアプリケーションが再開され，`onResume()` メソッドが呼び出される．再開処理はバックグラウンド状態のアプリケーションプロセス (Android OS において"キャッシュされたプロセス"と呼ばれる) を用い，プロセスの新規作成は行われないため，通常はプロセス無しの状態からの起動よりも再開による起動の方が短い時間でアプリケーションを起動できる．

3. 基礎性能評価

本章にて CMS GC と SS GC の性能評価を行う．

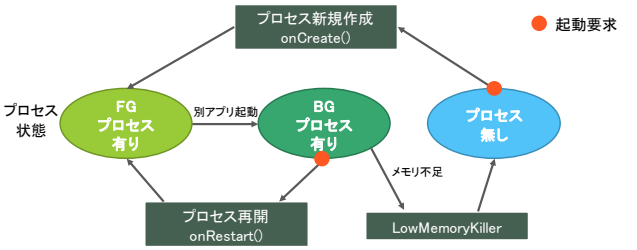

図 2. アプリケーションライフサイクル

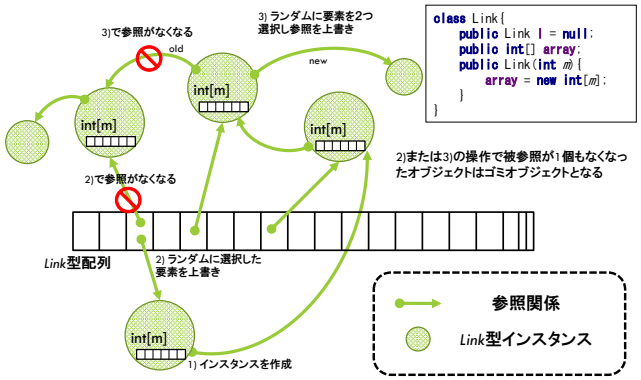

図 3. ベンチマーク概要

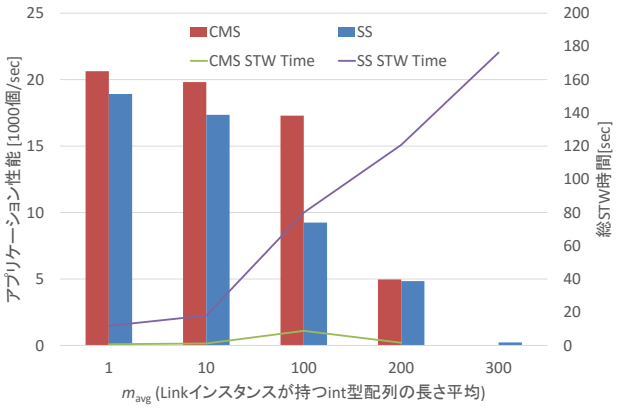

図 4. オブジェクトの大きさとアプリケーション性能

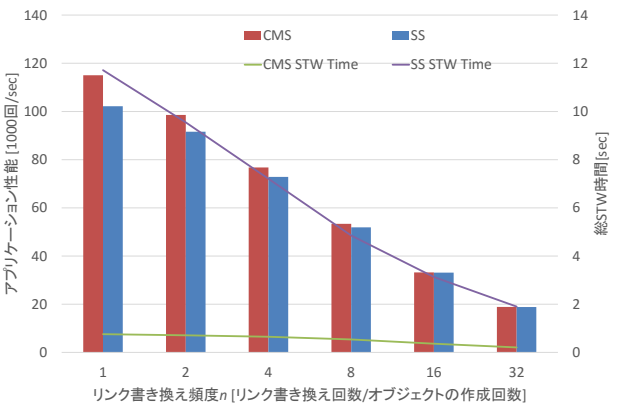

図 5. リンク書き換え頻度とアプリケーション性能

Vol.2016-GN-97 No.28
Vol.2016-CDS-15 No.28
Vol.2016-DCC-12 No.28
2016/1/22

3.1 ベンチマークアプリケーションによる評価

CMS GC と SS GC を用いて自作のベンチマークアプリケーションを実行した．図 3 にベンチマークアプリケーションの概要を示す．ベンチマークはまず，*Link* 型オブジェクトを参照する長さ 100,000 の *Link* 型配列を作成する．*Link* 型はベンチマーク用に作成したクラスであり，*Link* 型のインスタンスは長さ *m* の int 型配列と他の *Link* 型インスタンスへのポインタを 1 個持つ．*m* はインスタンスを作成する際に決定され，平均 m_{avg} の指数分布に従う乱数で決定される．インスタンスを 100,000 個作成し，配列の各要素がそれぞれを参照した状態から計測を始める．

ベンチマークは以下の 3 つの処理を繰り返す．1) *Link* 型インスタンスを 1 個作成する． 2) *Link* 型の配列の中からランダムに選択した要素を新たに作成したインスタンスで上書きする．3) *Link* 型配列からランダムに 2 つ要素を選択し，片方のインスタンスの中のポインタをもう片方へのポインタに上書きする．2)の結果，上書きされた要素から参照されていたインスタンスは配列からの参照を失う．この失われた参照が唯一の参照であった場合は，このインスタンスはゴミオブジェクトとなる．3)の書き換えは *n* 回行い，本稿ではこの *n* をリンク書き換え頻度と呼ぶ．2)および 3)の選択は，一様分布乱数により行う．測定時間は 3 分間行った．測定に使用した端末は表 1 の通りである．

表 1 使用端末

Device name	Nexus 7　(2013)
OS	Android 5.1.1
CPU	Qualcomm Snapdragon S4 Pro,1.5 GHz
Memory	2GB

測定結果を図 4，図 5 示す．各図の左縦軸はアプリケーション性能(オブジェクトの作成個数)であり縦棒で示されている．右縦軸は 3 分間の計測中に発生した STW 時間の合計で折れ線にて示されている．図 4 の横軸は *Link* 型オブジェクトのインスタンスが持つ int 型配列の長さの平均 m_{avg} である．また，m_{avg} とヒープサイズの関係を図 6 に示す．図 5 の横軸はリンク書き換え頻度 *n* を示している．図 4，図 5 より，CMS GC はアプリケーション性能と STW 時間の両方において，すべての場合において SS GC よりも優れていることが確認できる．また，m_{avg} が大きくなると SS GC の STW 時間が非常に大きくなることが確認できる．これは SS GC が GC 実行のたびに非ゴミオブジェクトをコピーするからだと考えられる．また，図 6 より本実験のように多くのメモリを消費するアプリケーションにおいては CMS GC より SS GC の方がヒープサイズが小さいことが確認できる．図 4 の m_{avg} が 300 の場合 CMS GC では Out of Memory Error が発生し計測が途中で終了してしまい，計測を正常に行うことができなかった．これに関しては次節で考察を行う．

3.2 メモリ確保成功率

前節において Out of Memory Error が発生して計測が最後

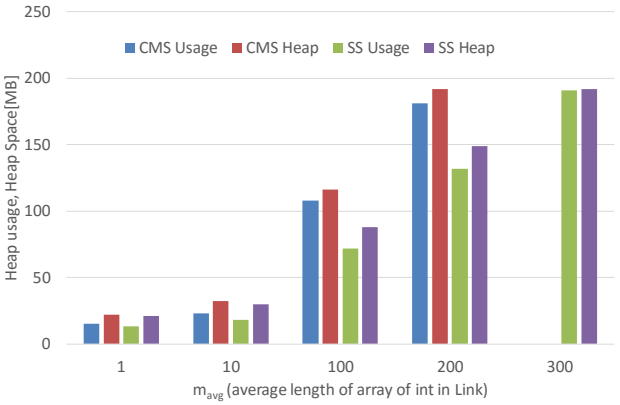

図 6. ヒープ使用量

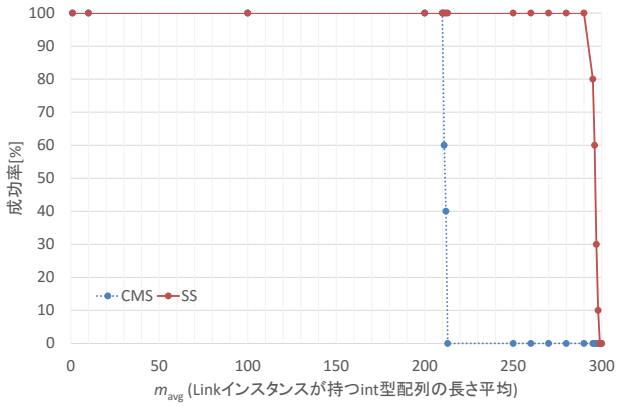

図 7. メモリ確保成功率

まで行えない場合があることを示した．本節では，オブジェクトの大きさとオブジェクト作成(メモリ確保)の成功率の関係を調査する．確認では，オブジェクトの作成を 1,000,000 回行いアプリケーションが Out of Memory Error を出さず正常に終了したら成功とした．この確認操作を 10 回行うことにより成功率を求めた．結果を図 7 に示す．図 7 の縦軸は成功率で，横軸が m_{avg} である．図より，SS GC のメモリ確保成功率は CMS GC より高いことが確認できる．

3.3 実アプリケーションによる評価

本節にて，実アプリケーションを用いての各 GC の評価を行う．評価実験の内容は以下の通りである．

まず，アプリケーションを①または②の状態にする．①アプリケーションを起動した状態，②アプリケーションを起動し規定の操作を行った状態．そして，この状態でアプ

Vol.2016-GN-97 No.28
Vol.2016-CDS-15 No.28
Vol.2016-DCC-12 No.28
2016/1/22

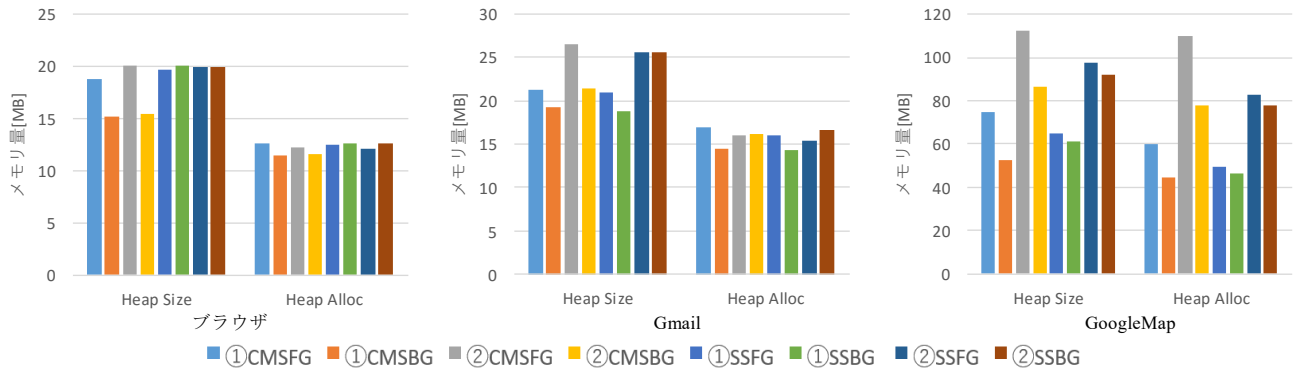

図 8. 実アプリケーションメモリ量

リケーションのメモリ使用量を調査する. これを「フォアグラウンド時メモリ」と呼ぶ. 次にホームボタンを押しアプリケーションをバックグラウンド状態にしてバックグラウンド時のメモリ使用量を調査する. これを「バックグラウンド時メモリ」と呼ぶ. 使用したアプリケーション名と行った操作は表 2 の通りである. アプリケーションの起動と操作は ADB コマンドを用いて人手を介さず自動的に実行し, 再現性を確保した.

測定結果を図 8 に示す. 図の縦軸は Heap Size, Heap Allloc のメモリ量で Heap Size は OS がアプリケーションプロセスに与えたヒープの量であり, OS の視点で使用中のメモリ量である. Heap Alloc は Heap Size 中にアプリケーションが実際にデータを格納して使用しているメモリ量であり, アプリケーションの視点で使用中のメモリである. 操作を与える前①と操作を与えた後②のメモリ量を比較するとブラウザは操作を与えても ART が管理しているメモリ量には変化が無いことがわかる. Gmail と GoogleMap は操作を与えた後のヒープサイズが大きい傾向があることが確認できる. CMS GC と SS GC のヒープサイズを比較するとフォアグラウンド時は SS GC のヒープサイズは CMS GC と比べて同等か低い値を示している. バックグラウンド時のヒープサイズを比較すると CMS GC が SS GC よりも低い値を示している. この結果から実際のアプリケーションにおいてはメモリ使用量の側面で CMS GC が SS GC より優れていることが分かる. ベンチマークアプリとの優劣差について次節で考察を行う.

表 2. アプリケーション名と操作

アプリ名	操作
ブラウザ	google.com をブラウザで開く. 検索フォームに"kogakuin"と入力し検索. 検索結果の中の工学院大学の Web ページのリンクをタップし工学院大学の Web ページを開く.
Gmail	メールを新規作成, 件名宛先アドレス, 本文を入力して送信を行う. 件名:title, 本文:body
GoogleMap	JR 山手線の駅を東京から神田まで外回りで順に検索し表示を行う.

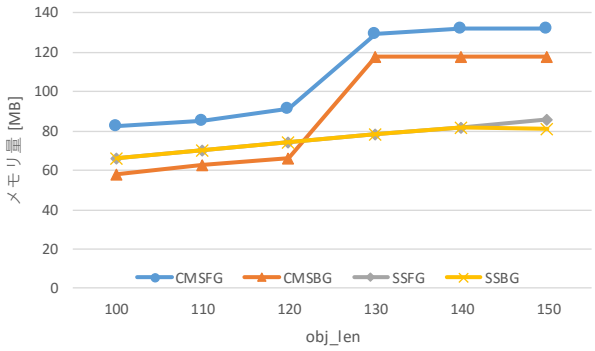

図 9. ベンチマークアプリ Heap Size

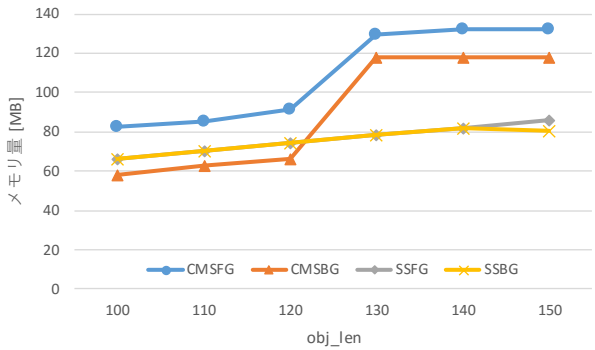

図 10. ベンチマークアプリ Heap Alloc

3.4 考察

メモリ使用量が大きいにおいてベンチマークアプリケーションでは SS GC の方が良い(メモリ使用量が少ない)が前節で計測した実際のアプリケーションにおいては CMS GC の方が SS GC より良い結果となった. 本節でベンチマークアプリケーションのメモリ使用量の詳細な考察を行う. 単純化のため, オブジェクトが持つ int 型配列の長さを乱数ではなく定数 ob j_len とした. そして, 発生させるオブジェクトの obj_len を 100 から 150 と変化させ obj_len とメモリ量の関係を測定した. 測定結果を図 9, 図 10 に示す. 各図は Heap Size, Heap Alloc をそれぞれ示し, 縦軸はメモリ量である. 図より SS GC の Heap Size と Heap Alloc は obj_len の値に対して線形的に増加していることが分かる. しかし, CMS GC の Heap Size と Heap Alloc は

Vol.2016-GN-97 No.28
Vol.2016-CDS-15 No.28
Vol.2016-DCC-12 No.28
2016/1/22

obj_len の 120 と 130 の値を境に急激に増えていることが確認できる．obj_len が 120，130 のときのメモリ割り当て状況を表 3 に示す．表 3 より，obj_len ＝120 の場合(長さ 120 の int 型配列を 100,000 個作成する場合)，CMS GC における 4byte array の Total Size が 47.7MB であり，アプリケーションの要求バイト数(4*120*100,000＝45.78 MB)とほぼ一致するが，obj_len ＝ 130 の場合は，4byte array の Total Size が 98.1MB でありアプリケーションの要求バイト数(4*130*100,000＝45.78 MB)を大きく上回っていることがわかる．これに対して SS GC を用いた場合は，4byte array の Total Size はアプリケーションの要求サイズを微小に上回る程度であり，obj_len＝130 の値に例においては CMS GC より消費メモリ量が大幅に少ないことが分かる．

表 3. 割り当てられたオブジェクト数と容量

obj_ave 120	CMS		SS	
	FG	BG	FG	BG
4-byte array 数[個]	100321	100356	100335	100359
4-byte array Total Size[MB]	47.713	47.714	51.524	51524
obj_ove 130	CMS		SS	
	FG	BG	FG	BG
4-byte array Count[num]	100322	100357	100335	100358
4-byte array Total Size[MB]	98.067	98.068	47.709	47.71

4. 提案手法

4.1 アプリケーション状態とオブジェクトの大きさに応じた GC 切り替え

前章にて Android OS に標準で使用されている CMS GC ではオブジェクトの大きさが一定値を超えると急激にメモリ使用量が増え非効率的な状態となることが確認できた．よって，このような状況では使用メモリ量の側面では SS GC を用いることが好ましいと言える．しかし，SS GC は STW 時間が長くアプリケーション性能や応答性の低下を招くと考えられる．そこで，大きなオブジェクトを発生させるアプリケーションでありそのアプリケーションがバックグラウンド状態に移行した場合に GC を SS GC に切り替えメモリ使用量を抑える手法を提案する．この手法を用いて本稿の実装では，バックグラウンド状態のアプリケーションのメモリ使用量を抑えることにより LowMemorryKiller によるプロセスキルの頻度が軽減されバックグラウンド状態のアプリケーションプロセスの保持数が増えることも期待できる．

4.2 提案手法の実装

本稿の実装では ART のアプリケーション実行時に GC のアルゴリズムを切り替える機能を用いてアプリケーションプロセスがフォアグラウンド状態，バックグラウンド状態で適用する GC アルゴリズムをそれぞれ設定してアプリケーションプロセスの状態に応じて使用される GC を切り替える．大きなメモリを使用するアプリケーションであるか否かの判定は，ベンチマークアプリケーションにおいて配列長が 125 を超えるか否かをもって決定した．

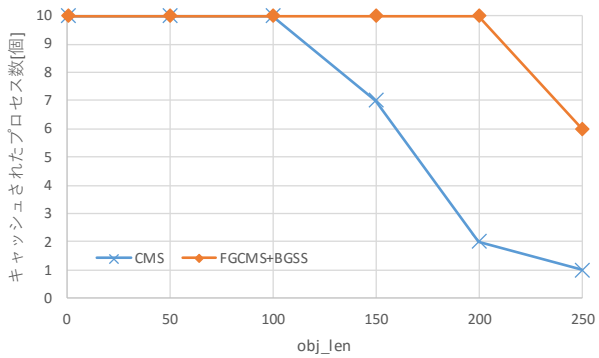

図 11. キャッシュされたプロセス数

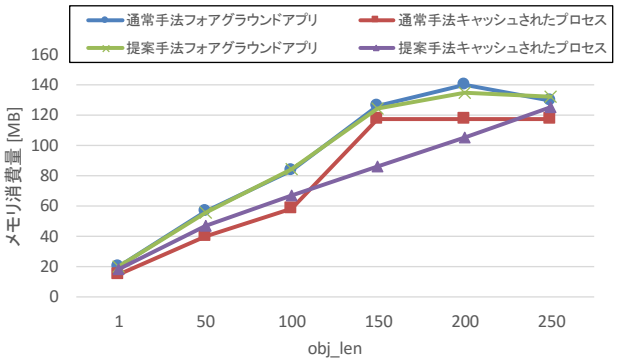

図 12. プロセスの各状態におけるメモリ消費量

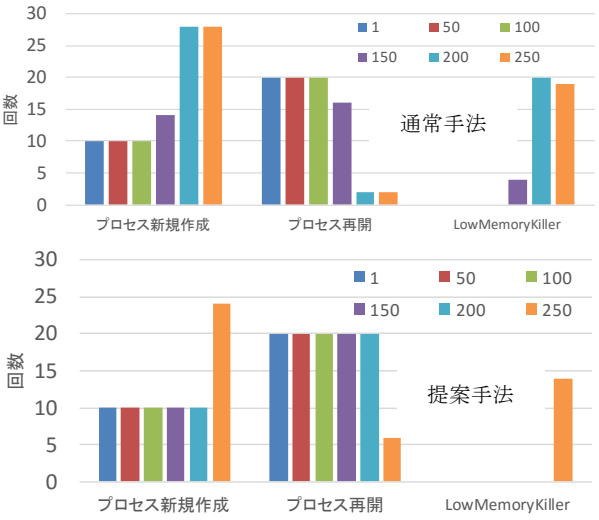

図 13. プロセス新規作成・再開，LMK 起動回数

5. 性能評価

本章にて提案手法の有効性を評価する．

5.1 測定方法

以下の測定を通常設定の Android OS(フォアグラウンド時 CMS GC，バックグラウンド時 SS GC)と，提案手法適用状態の Android OS(フォアグラウンド時 CMS GC，バックグラウンド時 SS GC)にて行い，それらのプロセスの消費メモリ量，アプリケーション起動時における再開成功確率，OS

内にキャッシュされているアプリケーションプロセスの数を評価した．実験内容は以下の通りである．

ベンチマークアプリケーションを 10 個(app00, app01, …, app09)用意し，それらを順に複数回起動する．起動順は，app00, app01, …, app09, app00, app01, ..., app09, app00, app01, ..., app09 である．すなわち，各アプリケーション 3 回ずつ起動する．そして，最終状態におけるキャッシュされたプロセスの数(バックグラウンドプロセスの数)と，最終状態における各アプリケーションプロセスの消費メモリ量，そして 30 回のアプリケーション起動のうち再開として起動された回数を求める．

起動したアプリケーション(app00～app09)は 3.4 節のベンチマークアプリケーションであり，長さ `obj_len` の int 型配列を含む非ゴミオブジェクトを 100,000 個保持する．アプリケーションは起動時にオブジェクトを 500,000 個発生させる．`obj_len` を変化させて発生させるオブジェクトの大きさを変えて測定を行った．

5.2 測定結果

測定結果を図 11-13 に示す．図 11 の縦軸はキャッシュされたプロセス数を示している．横軸は `obj_len` の値である．図 11 より，提案手法はアプリケーションの消費メモリ量が大きい場合に通常手法より多くのアプリケーションプロセスをキャッシュできていることがわかる．図 12 より，提案手法適用時にプロセスが消費するメモリの量は通常手法使用時量より小さく，これがキャッシュされたプロセスの数の改善につながっていることが分かる．図 13 より提案手法適用時の方が高い確率で既存プロセスの再利用によるアプリケーション再開を実現していることが分かる．これにより，アプリケーション起動時の応答性が改善し，ユーザのストレス軽減に繋げることができると考えられる．前述のように，提案手法はプロセスの消費メモリ量を削減させることによりキャッシュされたプロセスの数を増加させており，これがプロセス再開数の向上に繋がっていると言える．

6. おわりに

本稿では，Android OS の実行環境である ART に実装されている GC である CMS GC と SS GC について基礎性能調査を行った．大きなサイズのメモリを消費するベンチマークアプリを用いた基礎調査により，アプリケーション性能において CMS GC が優れ，メモリ使用量の削減において SS GC が優れていることを示した．実アプリケーションを用いた評価により，大きなメモリを使用しないアプリケーションにおいてはメモリ使用量においても CMS GC が優れることを示した．そして，CMS GC は一定のサイズを超えるオブジェクトを確保すると，メモリ使用効率が大幅に低下することを示した．

次にこれらの調査結果をもとに，メモリ消費量が大きいアプリケーションが多い状況ではアプリケーションがフォアグラウンド状態のときは CMS GC を用い，バックグランド状態になったときに SS GC を用いる手法を提案した．そして，キャッシュされたプロセスの数，プロセスの消費メモリ量，アプリケーション起動における再開成功確率を評価し，提案手法の有効性を示した．

今後は，より多くの実アプリケーションを用いての評価，大きいアプリケーションの判定の閾値に関する考察，CMS 実装に改変による消費メモリ量の改善に取り組んでいく予定である．

謝辞

本研究は JSPS 科研費 25280022, 26730040, 15H02696 の助成を受けたものである．

参考文献

[1]永田恭輔，中村優太，野村　駿，山口実靖, "Dalvik VM コンカレント GC の STW 時間の短縮に関する考察"，情報処理学会 第 76 回全国大会，6J-2

[2] 野村駿，中村優太，坂本寛和，山口実靖，" Android における LRU を用いた終了プロセスの選定"，情報処理学会論文誌コンシューマ・デバイス＆システム（CDS）, No. 5, Vol. 1, pp. 9-19