Vol.2017-UBI-54 No.24
Vol.2017-CDS-19 No.24
2017/5/26

スマートフォンアプリケーションの特性を考慮した 世代別 GC の Promote 条件に関する一考察

森竜佑[†1]　小口正人[†2]　山口実靖[†1]

概要：Android の仮想マシン Android Runtime (ART)の機能に，Garbage Collection (GC)がある．GC は空きメモリ量が減少すると自動的に不要なオブジェクトの開放を行う．ART に実装されている GC の一つに世代別 GC がある．
世代別 GC では，オブジェクトは新世代領域あるいは旧世代領域に属し，新世代領域内 GC で一定回数回収されなかったオブジェクトは旧世代領域へと Promote(昇進)する．オブジェクトの Promote 条件の最適値はアプリケーションごとに異なっていることが考えられる．本研究では実アプリケーションにおける世代別 GC の性能向上を目指した Promote 条件の最適値に関する考察を行う．

キーワード：Android，Android Runtime，Garbage Collection，世代別 GC

Promotion Condition Optimization based on Application Features in Generational GC of Smartphone Application Runtime

1. はじめに

スマートフォンやタブレット PC が普及し，これら端末の重要性が高まっている．Android はこれらの端末のプラットフォームとして高いシェアを持ち，2016 年第 2 四半期の世界市場における全スマートフォンの出荷台数における Android のシェアは 87.6%である．また，スマートフォン，タブレット PC 以外にも，腕時計，音楽プレイヤー，カーナビなど様々なデバイスで採用されており，重要性が高まっている．

Android には Low memory killer という機能があり，メモリ不足になるとプロセスを強制終了してしまう．同じプロクラムを再利用するときにプロセスの再生成が必要になり，遅い．よって，プロセスメモリサイズを小さくすることが重要である．

Android は ART（Android Runtime）上で動作する．

ART のメモリ管理機能の 1 つに GC があり，GC がメモリを解放する．ART の GC の動作を制御することにより，プロセスのメモリサイズを制御することができる．例えば Low memory killer の動作を抑制できる．

ART に実装されている GC の一つに世代別 GC がある．世代別 GC ではオブジェクトに年齢という概念を取り入れ，年齢ごとにオブジェクトを分類し GC を行う．ART における世代別 GC では，メモリ領域を新世代領域と旧世代領域に分類し，新世代領域内 GC で一定回数回収されなかったオブジェクトは旧世代領域へと Promote(昇進)する．また過去の研究にて，オブジェクトの分類の改善により GC の性能を向上できる可能性が示されており[1]，Promote 条件の制御による世代別 GC の性能向上に関する考察がベンチマークアプリケーションを用いて行われている[2]．本研究では，GC におけるスレッド停止時間（STW 時間）とメモリサイズがおおむねトレードオフの関係にあることを利用し，より少ない STW 時間の劣化でメモリサイズ低減を達成する手法について考察する．具体的には，世代別 GC の Promote 条件を厳しくし，メモリサイズ縮小させる手法を述べる．そして，オブジェクトの特性を考慮してこの手法を改善する手法を提案する．

2. Garbage Collection と GC アルゴリズム

2.1 Garbage Collection (GC)

GC は，メモリ領域に割り当てられたオブジェクトのうち使われなくなったオブジェクトを自動的に発見し開放する機能である．

図 1 の様にメモリ空間上にオブジェクトが配置されているとする．この中でどこからも参照できないオブジェクトがゴミオブジェクトとそれ以外のオブジェクトを非ゴミオブジェクトと呼ばれる．GC 処理を実行する前はオブジェクトが参照されているか否かの判別はできておらず，GC が参照を調査しゴミオブジェクトを発見する．

GC 処理が行われるとメモリ空間上の非ゴミオブジェクトにマークをつけていき，マークが付かなかったゴミオブジェクトを開放する．また，GC 処理中はアプリケーションスレッドが停止する．この現象は Stop The World（STW）と呼ばれる．

GC には複数のアルゴリズムがあり，次節以降に代表的なアルゴリズムを紹介する．

†1 工学院大学
Kogakuin University
†2 お茶の水女子大学
Ochanomizu University

Vol.2017-UBI-54 No.24
Vol.2017-CDS-19 No.24
2017/5/26

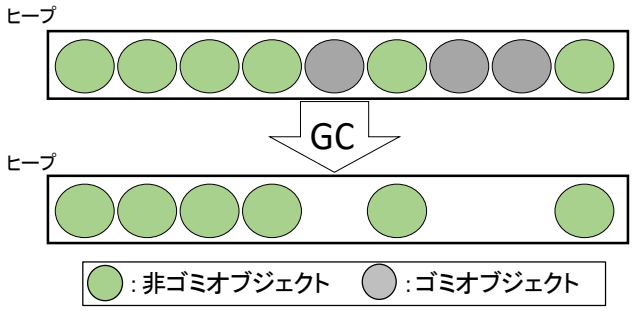

図 1 Garbage Collection

2.2 Mark and Sweep GC アルゴリズム

本節では Mark and Sweep GC アルゴリズムについて述べる．Mark and Sweep GC では，最初に，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．次にそれら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ていく．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，それらのメモリ領域は再度利用可能な状態になる．

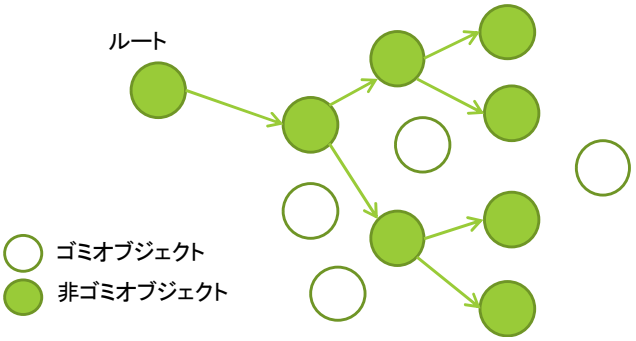

図 2 Mark and Sweep GC

2.3 Copying GC アルゴリズム

本節では Copying GC アルゴリズムについて述べる．Copying GC ではヒープを図 3 のように From 空間と To 空間の 2 つに分けて使用し，アプリケーションが新たなオブジェクトを生成すると，From 空間にそれらを割り当てる．

GC が実行されると前節で述べた Mark and Sweep GC と同様の方法で非ゴミオブジェクトを見つける．次に非ゴミオブジェクトを To 空間の端からコピーしていき，From 空間を空にする．最後に，From 空間と To 空間を入れ替える．Copying GC ではオブジェクトを To 空間にコピーする際にコンパクションと同じ処理が行われるため，ヒープが断片化することがない．

2.4 世代別 GC アルゴリズム

本節では，世代別 GC アルゴリズムについて述べる．世代別 GC とは，「生成されたばかりのオブジェクトはすぐに GC によって回収され，回収されずに長く生き残るオブジェクトは少ない」という経験に基づいた仮説をベースとしたアルゴリズムである．

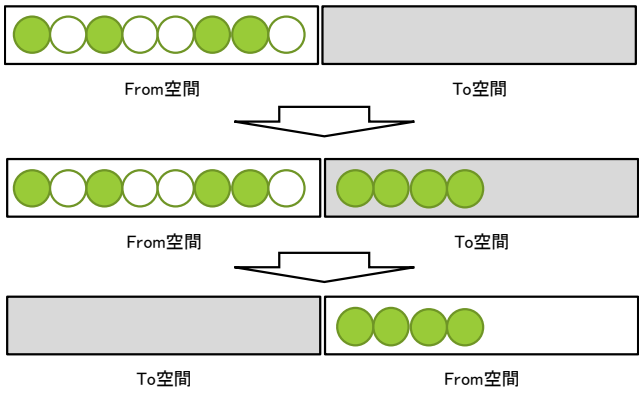

図 3 Copying GC

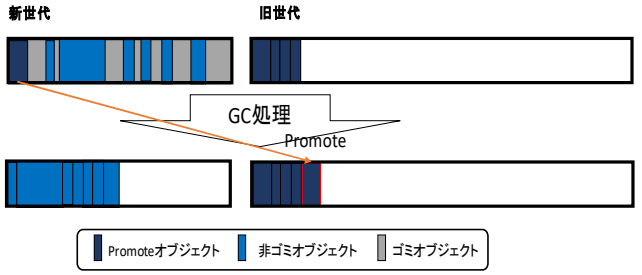

図 4 世代別 GC

わずかな時間で消滅する短命オブジェクトと長時間生き残る長寿オブジェクトを分けて管理し，GC 対象となるオブジェクトを限定することで GC の処理時間の削減が期待を図っている．図 4 のように新世代領域と旧世代領域に分けられ，新世代領域には生成されてから間もないオブジェクトが格納され，旧世代領域には生成されてから一定回数 GC を経験し回収されなかったオブジェクトが格納される．

アプリケーションがオブジェクトを生成すると，オブジェクトは新世代領域に割り当てられ，新世代領域がいっぱいになると新世代領域を対象とした GC が行われる．この GC によってゴミオブジェクトは回収され，一定回数生きた非ゴミオブジェクトは新世代領域から旧世代領域へ移動される．これを Promote（昇進）と呼ぶ．また，旧世代領域がいっぱいになると領域全体を対象とした GC を行い，ゴミオブジェクトを回収する．

2.5 ART における世代別 GC

ART には Generational Semi Space(GSS)という世代別 GC が実装されている．GSS ではオブジェクトを新世代領域に入るものと旧世代領域に入るものの 2 グループに分けて管理し，範囲の異なる 2 つの GC を行う．また，新世代領域用の GC としては前章で述べた Copying GC アルゴリズムを採用しており，From 空間と To 空間に分かれている．

Vol.2017-UBI-54 No.24
Vol.2017-CDS-19 No.24
2017/5/26

生成されたばかりのオブジェクトは新世代領域に置かれ，bpsGC(bump pointer space GC)によって高速にかつ積極的に回収される．bpsGC を 2 回生き残ったオブジェクトは旧世代領域へと Promote される．そして Promote したオブジェクトの累積バイト数が PromotedThreshold（初期設定にて4MB）を超えるたびに whole GC が行われる．whole GC の範囲は全領域であり GC 時間が長いため，頻繁に whole GC を行うことはアプリケーションの性能上は好ましくないと言える．ただし，旧世代領域に Promote してから参照されなくなったオブジェクトは bpsGC の調査範囲外となってしまい，whole GC を行うまで回収できなくなってしまうため，whole GC の回数の削減はヒープ拡大の原因になる可能性が考えられる．

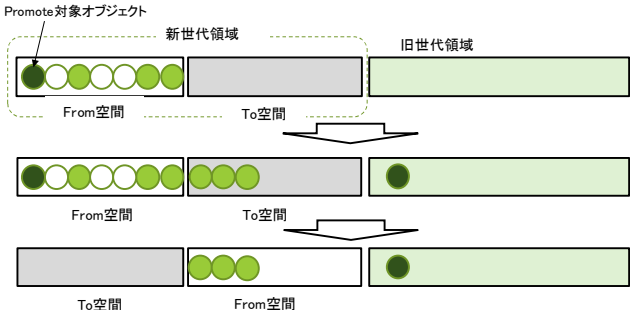

図 5 Generational Semi Space

3. スマートフォンアプリケーションにおけるオブジェクトの特性

本章にて，一般公開されている既存のアプリケーションにおける生成オブジェクトのサイズや寿命などの特性の傾向について述べる．

文献[3][4]にて，Android のアプリケーションにおいて生成されるオブジェクトの特性の調査結果が示されている．これらの文献では，ART の動作観察システムを構築し，アプリケーションランタイム内におけるオブジェクトの生成と GC による回収の観察を行い，特性ごとの寿命の傾向の調査を行っている．同文献の調査では， 2016 年 6 月 12 日における Google Play Store の無料アプリケーションの全カテゴリの 1 位のアプリケーションを調査用端末にインストールし，それらのアプリケーションにおけるオブジェクトの傾向を調査している．

同報告では，オブジェクトの寿命は短いものがもっとも多く，寿命が延びるに従い該当オブジェクトの数が単調に減少することが示されている．ただし，同報告における年齢はそのオブジェクトが経験した GC の回数であり，寿命とはそのオブジェクトが回収されるまでに経験した GC の回数の意味である．同調査にて，寿命が 0(最初の GC にて回収される)オブジェクトの数が最も多く，全体の約 80%を占めることが示されている．

オブジェクトサイズに関しては，32 バイト程度のものが最も多く，それ以上のサイズのオブジェクトの数は概ねサイズに対して単調に減少している．また，これらの傾向はアプリケーションに依存せず成立していたことが示されている．また，オブジェクトのサイズと寿命の関係に関しては，オブジェクトサイズ小さいほど寿命が 0 である確率が高く，平均寿命が短い傾向があるこが示されている．

これらの傾向より，オブジェクトがゴミになりやすいか否かを推定することが可能であり，GC 性能の向上が可能であると予想される．

4. Promote 抑制手法の提案

本章にて，ART の GSS の Promote 条件を厳しくし，ヒープサイズの削減を目指す手法を 2 つ提案する．一つはオブジェクトの特性を考慮せずに Promote を抑制する手法であり，本稿では単純 Promote 抑制手法と呼ぶ．もう一つは，オブジェクトのサイズを考慮して Promote を抑制する手法で有り，本稿ではオブジェクトサイズを考慮した Promote 抑制手法と呼ぶ．

4.1 オブジェクトサイズを考慮していない Promote 抑制手法

本節にて，オブジェクトサイズを考慮せずに Promote を抑制し，ヒープサイズの削減を目指す単純な Promote 抑制手法を提案する．

前述のように，ART の GSS GC は GC を 2 回経験すると旧世代領域に昇進する．本手法では，GC を 2 回以上経験したオブジェクトは確率 p で Promote させ，1-p で Promote させない．ある GC 実行にて Promote させないと判定されたオブジェクトも，次回以降の GC で再度判定の対象となり，同様に確率 p で Promote させ，確率 1-p で Promote させない．

4.2 オブジェクトサイズを考慮した Promote 抑制手法

本節にて，オブジェクトサイズを考慮した Promote 抑制手法を提案する．

本手法では，GC を 2 回以上経験したオブジェクトのうちサイズが m [byte]以下のオブジェクトは確率 p で Promote させ，確率 1-p で Promote させない様に制御する．サイズがそれ以上のオブジェクトは，通常通り GC を 2 経験すると必ず Promote させる．

Vol.2017-UBI-54 No.24
Vol.2017-CDS-19 No.24
2017/5/26

前章で述べたとおり，サイズが小さいオブジェクトは，また短命である確率が高いという傾向が確認されている．その傾向を用いて，サイズが小さなオブジェクトに対して Promote することを抑制することにより，近い将来にゴミになってしまうオブジェクオが旧世代領域に Promote してしまい GC による回収を長期間回避してしまう現象を減少できると期待できる．

5. 性能評価

本章にて，提案手法の性能評価を行う．

5.1 評価方法

性能評価には Google Map を用い，GC を起こさせそれを観察した．Google Map 起動中の処理は全て自動化し，次のように処理をした．山手線の駅を東京駅から外回りに検索していき，最後の神田駅を検索し画面に表示された時点で測定終了とした．また，ある駅と次の駅を検索するまでの間隔は 5 秒とした．前章で述べた，m と p の値はそれぞれ 16，0.5 に設定した． 1 回の GC で Promote するオブジェクトが少なくなることを考慮し，PromotedThreshold を 1 MB に変更した．測定に用いた端末は Nexus7 であり，仕様は表 1 のとおりである．

表 1 測定環境

端末名	Nexus7(2013)
OS	Android 6.0.1（改変済み）
CPU	Snapdragon S4 Pro 1.5GHz
メモリ	2GB

5.2 評価結果

図 6 に各 GC 直後のヒープサイズを示す．図 6 の横軸は GC 回数を示している．結果より，オブジェクトサイズを考慮した Promote 抑制手法と単純 Promote 抑制手法いずれも通常手法と比較してヒープサイズを縮小できていることを確認できる．また，オブジェクトサイズを考慮した提案手法と考慮しない提案手法は同程度の結果となった．これらの平均を図 7 に示す．同図からも，両提案手法によりも 1 つのサイズが低減されていることがわかる．

図 8 に各 GC 処理中に起こった STW 時間を示す．図より，いずれの提案手法も通常手法より STW 時間の増加が確認できる．また，オブジェクトサイズを考慮した Promote 抑制手法と単純 Promote 抑制手法，では前者の方が STW 時間の増加が少ないことがわかる．この平均を図 9 に示す．

図 6 から 9 より，提案手法により STW 時間を増加させつつも，メモリサイズを低減させることが可能であることがわかる．また，オブジェクトサイズを考慮することにより，より小さな STW 時間増加で同程度のメモリ量低減を

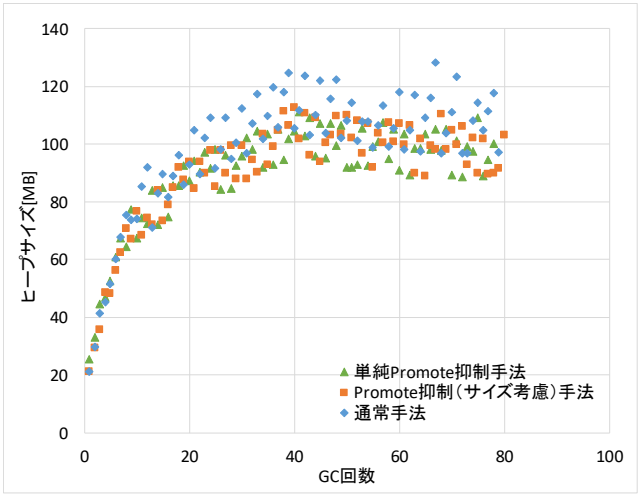

図 6 ヒープサイズ

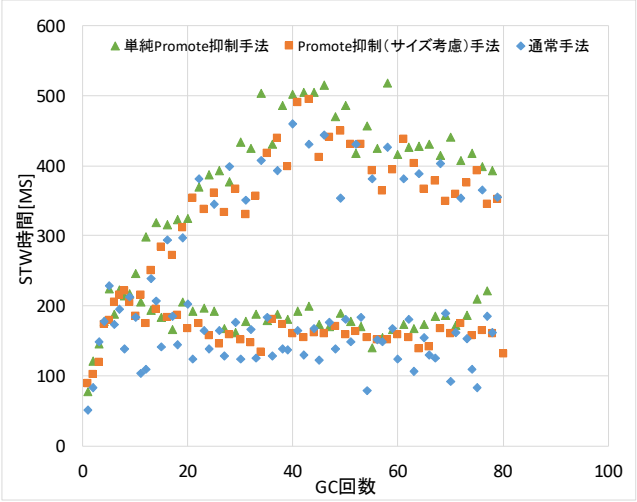

図 7 STW 時間

実現できていることがわかる．

図 6 の結果同様，オブジェクトサイズを考慮した提案手法と考慮しない提案手法いずれも通常手法と比較してヒープサイズの縮小を確認でき，またオブジェクトサイズを考慮した提案手法と考慮しない提案手法はほぼ同じ結果であることがわかる．また，図 7 の結果同様，通常手法と比較しいずれの提案手法も STW 時間が増加していることがわかる．

6. 考察

6.1 STW 時間の増加

前章の性能評価にて，通常手法と比較しヒープサイズは縮小したが，STW 時間が増加してしまうことが確認された．本手法はオブジェクトサイズが小さいほど寿命が 0 である確率が高く，平均寿命が短い傾向を利用し，Promote してもすぐにゴミになる確率が高いと予測することで Promote 抑制させている．しかし，必ずしも Promote 抑制させたオブジェクトが次の GC によって回収されるとは限らず，予測の精度は必ずしも高いとは限らない．Promote 抑制されたオブジェクトがつぎの GC までに全てゴミとなっていれ

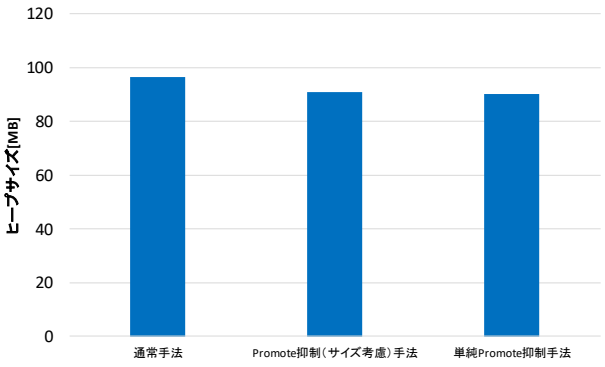

図 8 平均ヒープサイズ

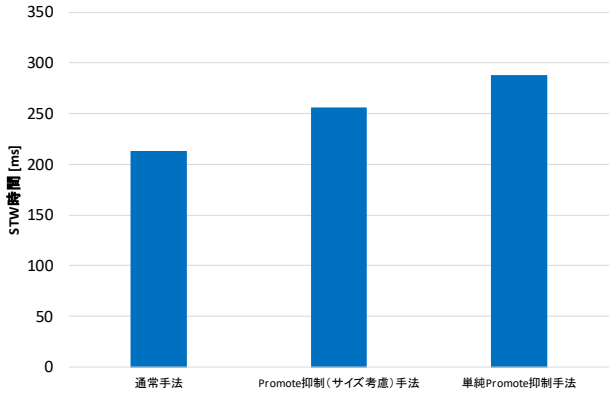

図 9 平均 STW 時間

ば，STW の増加は少ないと考えられる．Promote 抑制されたオブジェクトが次の GC でも生き残こる場合，その分だけ bps GC の参照されているオブジェクトへのマーク処理やコピー処理を行わなくてはいけなくなり，結果として STW が増加したと考えている．

STW 時間が増加する問題は，オブジェクト寿命の推定の精度を向上させることにより，より軽減できると期待できる．図 9 においても，相対的に精度が低い単純な手法より，より精度が高いオブジェクトサイズを考慮する手法の方が STW の増加が少なくなっており，この様に期待できることが分かる．文献[3][4]においては，オブジェクトサイズ以外の特性とオブジェクト寿命の関係も示されており，また文献[森竜全国]においてはオブジェクト作成時刻(アプリケーション開始時から経過時間)と寿命の関係も示されており，これらの特性も考慮して推定を改善させることにより STW 時間をより短くできると期待できる．

6.2 使用目的に応じた手法の選択

Android OS 搭載端末は小容量の RAM しか搭載していない端末も少なくない．また，本研究で測定のために用いた Google Map のように使用メモリ量の大きいアプリケーションが存在する．複数のアプリケーションを起動させ空きメモリ量が少なくなると，LowMemoryKiller が起動され，プロセスが強制終了される．前述の通り，この LowMemoryKiller の起動は好ましくなく，ユーザーのエクスペリエンスを低下させると思われる．

ユーザエクスペリエンスの改善方法として，STW 時間の増加の影響とメモリサイズの増加(および LowMemoryKiller 回数の増加)の家強を考慮して，アプリケーションごとに GC のポリシーを選択する手法が考えられる．

すなわち，メモリ消費量が大きいが高いインタラクティブ性を要求しないアプリケーションに対してはメモリサイズ削減を優先した GC を適用し，高いインタラクティブ性を要求するアプリケーションに対しては STW 時間の短縮を優先した GC を選択することにより，より良い使用性を提供できると考えられる．前者の例としては Web ブラウザが，後者の例としてはエンターテインメント(ゲームや動画再生)などが考えられる．

7. 関連研究

本章で GC に関する研究と Android の GC やメモリ管理についての研究を紹介する．

GC の研究については Java VM は世代別 GC の採用が主流であるが，GC のリアルタイム性を重視し，これを GC のインクリメンタル化し，リアルタイム性を向上させた研究[5]やスナップショット方式を採用することで GC リアルタイム化を実現する研究[6]がある．他にも，GC のリアルタイム性に関する研究 [7][8][9]，GC の並列処理化に関する研究 [10][11][12][13]などがある．

Android の GC に関する研究は旧バージョンの仮想マシン Davik VM に搭載されている GC の評価を行った研究[2][14]がある．[2]の研究はアプリケーションが発生するオブジェクト数とアプリケーション性能の関係，また，オブジェクト間の参照関係の書き換え頻度とアプリケーション性能の関係の調査を行っており，アプリケーションが生成するオブジェクト数と参照書き換え頻度が大きくなることで GC の STW 時間が大きくなることが報告されている．[14]の研究は Android を組込み機器に適用するため特定アプリケーションの実行優先度をリアルタイム優先度に設定して起動する仕組みを構築している．しかし，この仕組を利用した場合でも GC にアプリケーションが GC の影響を受けることは避けられないという．このように様々な観点から GC 性能の調査が行われているが，これらの調査はオブジェクト寿命を考慮していない．

Dalvik VM の性能向上に関しての研究[15][16]がある．[15]の研究はオブジェクトに再帰的にマークを行う処理に長い時間を要し，その要因に既にマーク済みのオブジェクトに対しても繰り返しマーク処理を行っていることを明らかにした．そして，マーク済みのオブジェクトを管理するための専用表をプロセッサに追加し，オブジェクトの探索時にこの表を参照することで，マーク済みのオブジェクト

に対する冗長なマーク処理を短縮する手法を提案し，GC の高速化を実現している．[16]の研究は STW 時間の短縮を目的として，オブジェクトのマーク中に生じたポインタの更新があったオブジェクトを再びマークするリマークは通常 STW で行われるが, リマークをコンカレントに実行し，リマーク中のポインタの更新があったオブジェクトをマークするリリマークを行う手法を提案し, STW 時間の短縮を実現している．両手法は本研究とは異なるアプローチで高速化を実現しており，このような手法と組み合わせることが好ましいといえる．

ART に実装されている GC アルゴリズムをアプリケーション状態によって切り替える研究[17]がある．この研究では ART に実装されている CMS GC と SS GC の 2 種類の GC の性能評価を行い, アプリケーション性能の観点では CMS GC，メモリ使用量の観点では SS GC が優れていることを明らかにしている．そして，アプリケーションがフォアグラウンド状態においては CMS GC，バックグラウンド状態においては SS GC に GC アルゴリズムを切り替えることにより，性能を落とさず端末全体のメモリ使用量を削減している．そして，より多くのアプリケーションキャッシュを保持することにより，アプリケーション再利用の際の起動時間の短縮を実現している．これに関連して，強制終了するアプリケーションプロセスの選定手法の改善し，ユーザが近い将来利用するアプリケーションのキャッシュを効率的に残しアプリケーション再利用時間の短縮を行った研究がある[18]．Android には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強制終了する Low Memory Killer という機能があり，このアプリケーションプロセスの選定手法に LRU を用いる手法を提案している．実際のユーザが使用したアプリケーション履歴に基づき評価を行った結果, LRU を用いると優れた性能を示すことが確認されている．本稿の手法において，メモリ使用量の削減が確認されたことからアプリケーションキャッシュにどのような影響を与えるかを調査することが重要であるといえる．

8. おわりに

本稿では，オブジェクトのサイズを考慮した Promote 抑制手法を提案し，Google Map による性能評価を行った．その結果，提案手法によりアプリケーションのヒープサイズを縮小させることができた．また，オブジェクトサイズを考慮することにより STW 時間を低減できることが確認された．

今後は別のアプリケーションでの性能調査と, STW の低減方法に関する考察を行っていく予定である．

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．

本研究は、JST、CREST の支援を受けたものである．

参考文献

[1] 濱中 真太郎 栗原 駿 福田 翔貴 小口 正人 山口 実靖,” Android アプリケーションのオブジェクトと GC の関係に関する一考察”，FIT2016，2016‐09
[2] 森 竜佑 濱中 真太郎 小口 正人 山口 実靖，” Promote 条件の制御による Android 世代別 GC の性能向上に関する一考察”， 情報処理学会第 79 回全国大会，2017-3
[3] Shintaro Hamanaka，Shun Kurihara，Shoki Fukuda， Masato Oguchi，Saneyasu Yamaguchi，” A Study on Object Lifetime in GC of Android Applications” , CANDAR 2016 ,2016-11
[4] Shintaro Hamanaka，Shun Kurihara，Shoki Fukuda，Ryusuke Mori， Masato Oguchi，Saneyasu Yamaguchi，” Object Lifetime Trend of Modern Android Applications for GC Performance Improvement” , ACM IMCOM 2017 , 2017-01
[5]湯浅太一，「実時間ごみ集め」，情報処理学会, Vol.35, No.11, pp.1006-1013, 1994
[6]遠藤 匠, 田中 陽, 前田 敦司, 山口 喜教, 「スナップショット GC による Java の GC のリアルタイム化」, 情報処理学会研究報告. SLDM, [システム LSI 設計技術] 2003(28), 41-46, 2003
[7]H. G. Baker. List Processing in Real-Time on a Serial Computer. Communications of the ACM, 21(4):280{294, 1978.
[8]H. G. Baker. The Treadmill: Real-Time Garbage Collection Without Motion Sickness. ACM SIGPLAN Notices, 27(3):66{70, March 1992.
[9]H. Lieberman and C. E. Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. Communications of the ACM, 26(6):419{429, 1983.
[10]H. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, pages 157{164, Toronto, Canada, June 1991.
[11]James O'Toole, Scott Nettles, and David Gi_ord. Concurrent compacting garbage collection of a persistent heap. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 161{174, Asheville, NC (USA), December 1993.
[12]Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M.Ste_ens. On-the-fly garbage collection: An exercise in cooperation. CACM, 21(11):966{975, November 1978.
[13]D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 113{123, New York, NY, 1993. ACM.
[14]東山知彦, 増田大樹, and 落合真一. "Android の応答性評価." 第 75 回全国大会講演論文集 2013.1 (2013): 35-36.
[15]河村 慎二, 津邑 公暁, “GC における冗長なマーク処理抑制のためのハードウェア支援手法”, 第 213 回システム・アーキテクチャ研究発表会（SWoPP2016）
[16]濱中真太郎・中村優太・野村駿・山口実靖, "DalvikVM コンカレント GC の STW 時間の短縮に関する一考察", 工学院大学研究報告第 117 号, pp. 33-38
[17]濱中真太郎，坂本寛和，小口正人，山口実靖, “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”, 第 97 回 GN・第 15 回 CDS・第 12 回 DCC 合同研究発表会, CDS-19
[18]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.