Vol.2014-GN-90 No.19
Vol.2014-CDS-9 No.19
Vol.2014-DCC-6 No.19
2014/1/24

Dalvik VM コンカレント GC の STW 時間の短縮に関する一考察

永田恭輔[†1]　中村優太[†1]　野村駿[†1]　山口実靖[†1]

スマートフォンやタブレット PC が普及し、これらの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、特に重要なプラットフォームとなっている。 Android には Dalvik VM という仮想機械が搭載されており、Android のアプリケーションは Dalvik VM の上で動作する。 Dalvik VM には GC 機能が搭載されており、この性能がアプリケーション性能に影響を与える。本稿では, Dalvik VM のコンカレント GC の STW 時間に着目し,非ゴミオブジェクト数や書き換え頻度と GC の各フェーズの処理時間の関係の調査を行った。そして、これらの測定をもとに STW 時間の短縮（ミューテータの性能低下抑制）手法に関する考察を行った。

Android operating system has Dalvik virtual machine and Androidapplications run on this virtual machine. Dalvik virtual machine hasGC (garbage collection) function, thus application developers do nothave to release memory by themselves. However, this GC functionsuspends application, which is called STW, during garbage collection.In this paper, we evaluate STW time of Dalvik GC and show that it isnot enough short in some cases. After the evaluation, we discuss amethod for decreasing STW time, with which remark process is executedconcurrently.

A Study on Shortening STW Time of Concurrent GC of Dalvik VM

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

1. はじめに

スマートフォンやタブレット PC が普及し、これらの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、特に重要なプラットフォームとなっている。Android には Dalvik VM という仮想機械が搭載されており、Android のアプリケーションは Dalvik VM の上で動作する。Dalvik VM には GC 機能が搭載されているため、アプリケーション開発者はメモリ開放を自ら行う必要がない。しかし、GC が動作するとアプリケーション（ミューテータ）の停止あるいは性能低下が生じるため、リアルタイム性の高いアプリケーションにおいては GC の起動が大きな問題となり、GC の起動を回避する様な開発が要求されることがある。本論文では、GC のストップザワールド（STW）時間に着目し、この時間を短縮する手法について考察する。STW 時間とは GC 処理中に完全にミューテータが停止してしまう時間のことであり、ミューテータとは GC の対象であるアプリケーションスレッドのことを指す。具体的には、Dalvik VM GC によるミューテータの停止時間の評価を行い、GC がミューテータ性能に大きな影響を与える条件を調査する。そして、GC の影響の提言手法について考察する。

2. Dalvik VM GC

Dalvik VM は GC アルゴリズムとしてマーク&スイープを採用している。マーク&スイープの利点として、実装が容易であること、参照カウントと異なり GC が動いてないときのオーバーヘッドが小さいこと、循環参照も解放できることが挙げられる。一方、欠点としては、全てのオブジェクトを辿らなければならないため時間がかかることが挙げられる。図 1 にマーク&スイープの動作を示す。マーク&スイープはマークフェーズとスイープフェーズに分かれており、マークフェーズでは非ゴミ（使用中）オブジェクトにマークを付け、スイープフェーズではマークの付いていないオブジェクト（ゴミオブジェクト、不使用オブジェクト）の回収を行う。

マークフェーズでは、最初に VM から直接参照されているオブジェクトをルートオブジェクトとしてマークする。次に、ルートオブジェクトから直接的あるいは間接的に参照されているオブジェクトを再帰的にリンクを辿ることによりマークする。これらのルートオブジェクトから参照可能なオブジェクトが非ゴミ（使用中）オブジェクトとなる。次に、スイープフェーズでは前述のマークフェーズにてマークされなかったオブジェクトをゴミ（不使用）オブジェクトとして回収する。また Dalvik VM の GC では、CoW にてプロセス間で共有されているメモリにマーキングによる書き込みが生じないように、ビットマップマーキング方式が採用されている[1]。

Android 2.2 以前の Dalvik VM の実装では、GC 処理とミューテータを並列に動作させることができず（以下これをノンコンカレント GC と呼ぶ）、GC 処理全体が STW 処理となっている。このため、GC によりアプリケーションが 100ms 以上停止することがしばしば発生すると指摘されており、この時間は許容範囲内ではないとの主張もなされている[2]。ノンコンカレント GC における GC 処理とミューテータ（アプリケーション）処理を図 1(a)に示す。一方、Android 2.3 以降の Dalvik VM の実装では、コンカレント GC が採用されており、図 2(b)の様に GC 処理の一部とミューテータを並列して動作させることが可能である。コンカレント GC の処理は以下の通りである。まず、イニシャルマーク処理としてルートオブジェクトに印をつける。この間は、ミューテータは停止（STW）する。次に、コンカレントマーク処理としてルートオブジェクトから辿れるオ

†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

Vol.2014-GN-90 No.19
Vol.2014-CDS-9 No.19
Vol.2014-DCC-6 No.19
2014/1/24

ブジェクトにマークしていく。この間は、ミューテータと GC が並列に動作し、ミューテータは停止しない。次に、リマーク処理としてコンカレントマーク処理中に生じた書き込みの整合性をとるために、ミューテータを停止（STW）させて変更が生じたオブジェクトを基にマークを行う。最後に、コンカレントスイープ処理としてマークの付いていないオブジェクトをゴミオブジェクトとみなし、フリーリストに繋ぎ、再利用可能な領域とする。コンカレントスイープ処理はミューテータと並列に処理が行われる。非コンカレント GC では全てのマーク処理中ミューテータが停止（STW）するのに対し、コンカレント GC では 2 度の停止（STW）の間ミューテータが動作するため、停止（STW）時間が大幅に削減される。

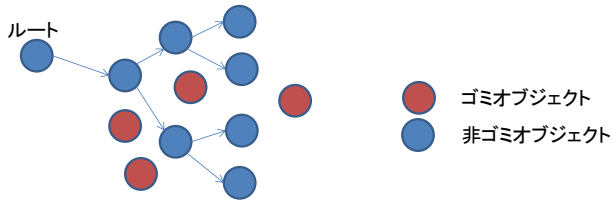

図 1. M&S GC のマークフェーズ終了時
オブジェクトの様子

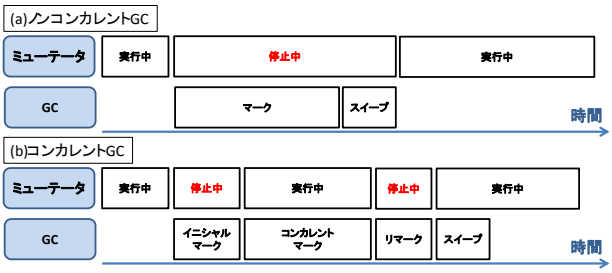

図 2. GC の処理

3. 基本性能調査

本章で Dalvik VM GC の性能の評価を行う。

3.1 測定環境

Nexus7 上で自作のベンチマークを実行し、GC の各処理の時間を測定した。端末の使用は表 1 の通りである。

表 1. 測定環境

OS	Android 4.1.2
CPU	NVIDIA Tegra 3 T30L 1.3GHz クアッドコア
メモリ	1GB
ストレージ	16GB

3.2 ベンチマーク

本節で使用したベンチマークについて説明する。ベンチマークは初めに、指定数のオブジェクト（これは非ゴミオブジェクトとなる）を生成する。次に、ゴミオブジェクトを生成し続け、GC を起動させる。非ゴミオブジェクトは常にルートオブジェクトから直接参照されており、ゴミオブジェクトとなることはない。また、非ゴミオブジェクト同士でランダムにリンクが張られており、このリンクはベンチマーク実行中に変更が行われる。リンクの変更に関しては、次節で説明する。非ゴミオブジェクト（ルートオブジェクトから辿れるオブジェクト）数やオブジェクト同士のリンクの書き換え数を変化させ、GC の処理にどのような影響を与えるかの評価を行う。また、GC の各処理時間の測定のために、Dalvik VM GC の実装に時刻取得関数を追加し、後述の Dirty カード量の測定のために、Dirty カード数をカウントする機能を追加した。

測定では GC 時間と GC によるミューテータの停止相当時間を測定している。GC 時間は GC のそれぞれのフェーズに要する時間である。ミューテータの停止相当時間とは GC により低下した性能をミューテータ停止に換算したものであり、図 3 の失われた性能を非 GC 中の性能で割った値である。図 3 は GC 前、GC 中、GC 後のミューテータの性能をモデル化したものであり、GC が稼働していない期間は α の性能が得られており、GC 稼働中は得られる性能が α より低いことを表している。図内の色つきの面積が GC により失われた処理量を表しており、これは GC が発生しなかったと仮定したとき（α の性能が得られ続けたと仮定したとき）に達成できた処理量と、GC 発生時に実際に達成された処理量の差である。本ベンチマークでは、性能とは単位時間あたりに実行できる所定の処理の回数であり、所定の処理とは、ゴミオブジェクト 1 個の生成と、非ゴミオブジェクト同士のリンクの書き換えで構成されている。

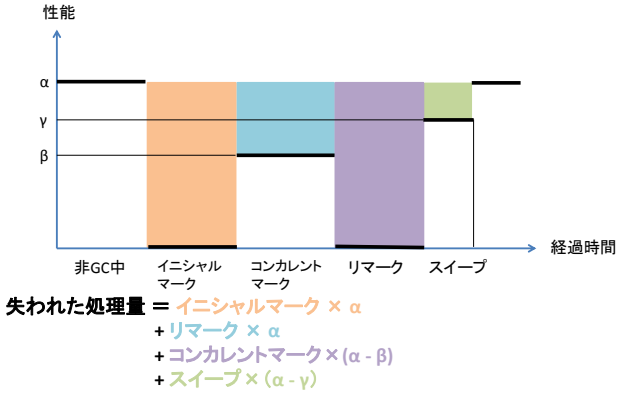

図 3. 失われた処理量の計算方法

3.3 リンク書き換え頻度

本節でリンク書き換え頻度について説明する。各非ゴミオブジェクトは、一様分布乱数でランダムに選択した非ゴミオブジェクトに対するリンクを 1 個保持しており、ベンチマーク実行中はゴミオブジェクト 1 個生成するたびに、ランダム選択された n 個のオブジェクトのリンクを別のものに変更する。この n が書き換え頻度であり、書き換え頻度 n の時は、ゴミオブジェクト 1 個生成するたびに n 個のオブジェクトのリンクが変更されることになる。コンカレントマーク中にリンク書き換えが行われたオブジェクトは、リマークにおける再調査の対象となる。

Vol.2014-GN-90 No.19
Vol.2014-CDS-9 No.19
Vol.2014-DCC-6 No.19
2014/1/24

3.4 非ゴミオブジェクトと GC 時間の関係

本節では、コンカレントマーク中にオブジェクトに対する変更が行われていない環境における非ゴミオブジェクト数と GC のそれぞれのフェーズの時間の関係について述べる。測定結果を図 4, 5, 6, 7, 8 に示す。すべての測定において書き換え頻度は 0 である。図 4 から図 8 より、非ゴミオブジェクト数が 10^3 以下では GC 時間の増加は小さいが、それ以上では非ゴミオブジェクト数が多いほど GC 時間が増加することが分かる。また、STW 時間（イニシャルマーク時間、リマーク時間）に着目すると、最長の例でも 6.6ms であることが分かる。

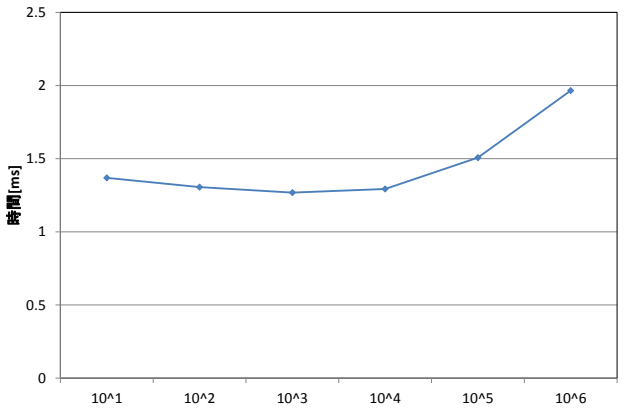

図 4. 非ゴミオブジェクト数とイニシャルマーク時間

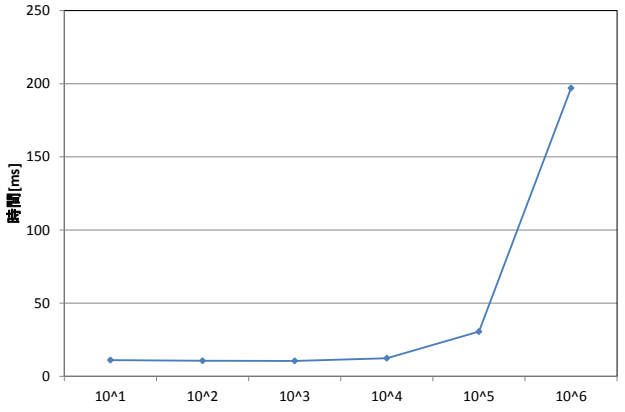

図 5. 非ゴミオブジェクト数とコンカレントマーク時間

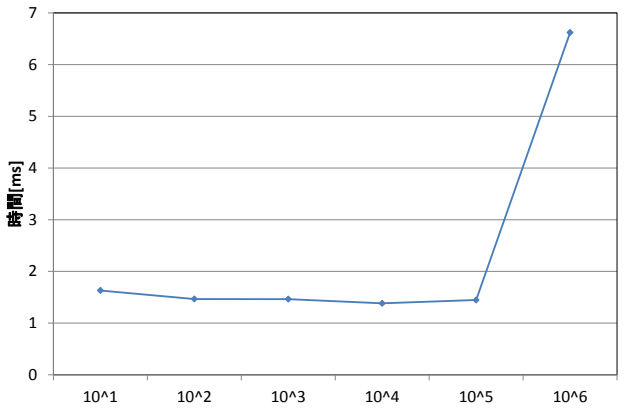

図 6. 非ゴミオブジェクト数とリマーク時間

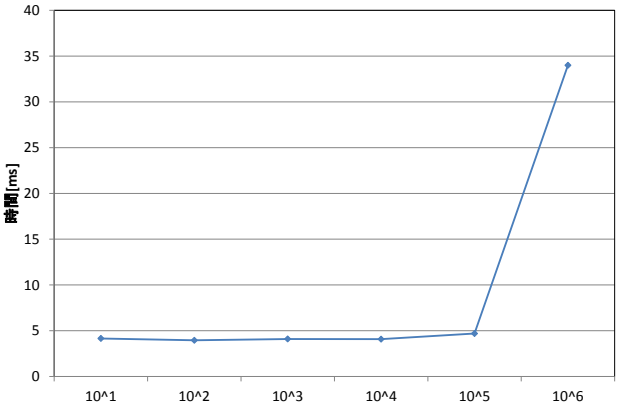

図 7. 非ゴミオブジェクト数とスイープ時間

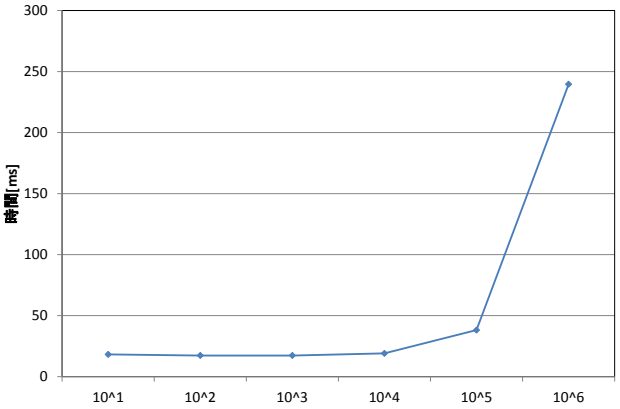

図 8. 非ゴミオブジェクト数と総 GC 時間

3.5 リンク書き換え頻度と GC 時間の関係

本節では、このリンク書き換え頻度を 1, 2, 4, 8, 16 と変化させたときのそれぞれのフェーズの時間について述べる。全ての測定において、ベンチマークの非ゴミオブジェクト数は 10^6 個である。測定結果を図 8, 9, 10, 11, 12, 13 に示す。図 11 より、書き換え頻度を増加させるにしたがって、リマーク時間が増加することが確認できる。これは、コンカレントマーク中に変更されたオブジェクトが増えるに従い、リマークで調査しなおすオブジェクト数が増えるためだと考えられる。STW 時間に着目すると、最長の例においては 50ms を越えていることが分かる。これは、一般的フレームレート（25fps や 30fps）におけるフレーム間の時間より大きく、人間もミューテータの停止を認知できる時間であると予想される。

Vol.2014-GN-90 No.19
Vol.2014-CDS-9 No.19
Vol.2014-DCC-6 No.19
2014/1/24

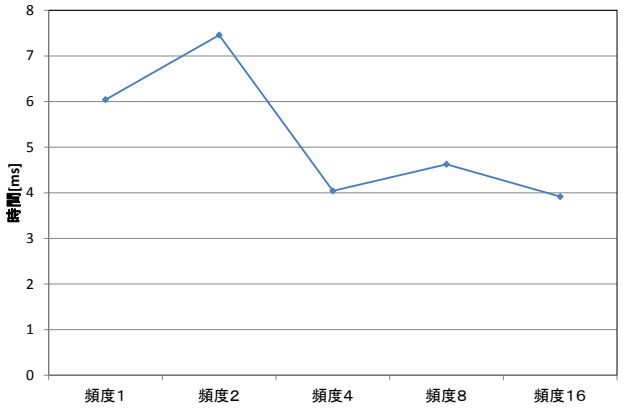

図 9. 書き換え頻度とイニシャルマーク時間

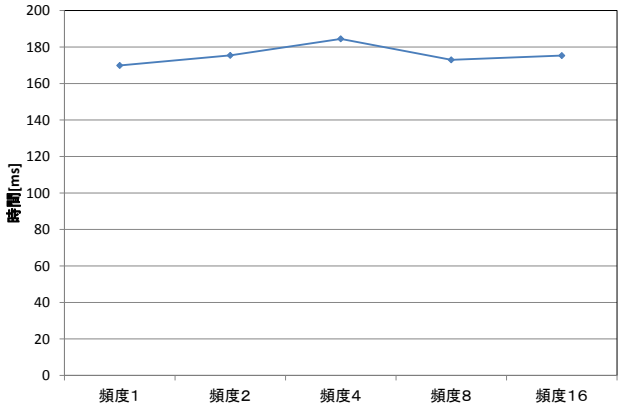

図 10. 書き換え頻度とコンカレントマーク時間

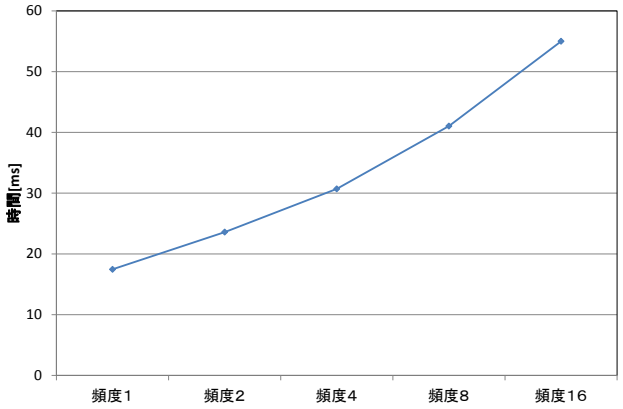

図 11. 書き換え頻度とリマーク時間

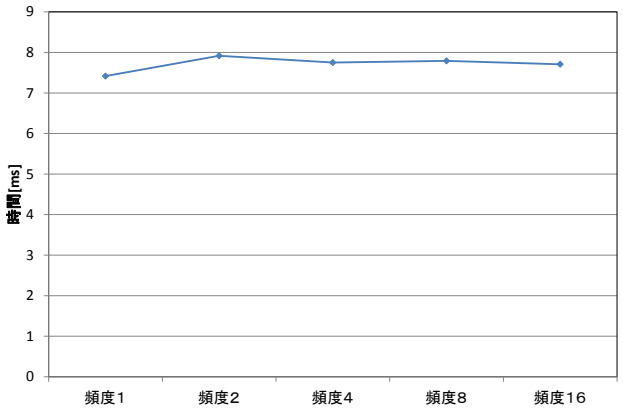

図 12. 書き換え頻度とスイープ時間

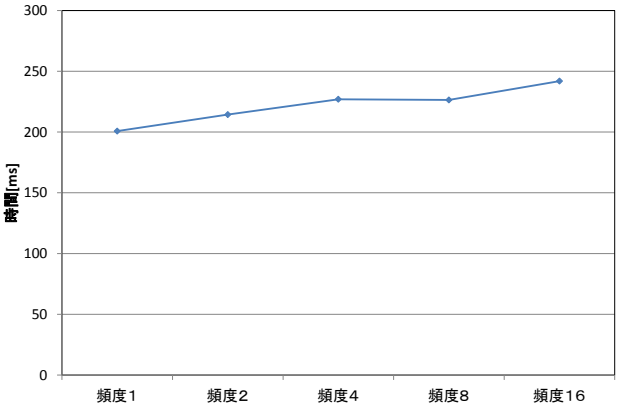

図 13. 書き換え頻度と総 GC 時間

3.6 ミューテータ停止相当時間

これにより、GC よるミューテータの性能低下を知ることができる。

本節では、非ゴミオブジェクト数とリンク書き換え頻度のそれぞれの変化による停止相当時間の変化について述べる。測定結果を図 14, 15 に示す。図 14 から、非ゴミオブジェクト数が増加しても、ミューテータの性能低下は少ないことが分かる。図 15 から、書き換え頻度が増加すると、性能低下が大きくなることが確認できる。

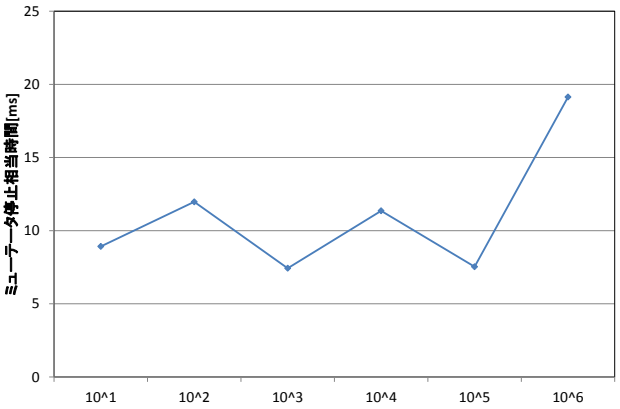

図 14. 非ゴミオブジェクト数と停止相当時間

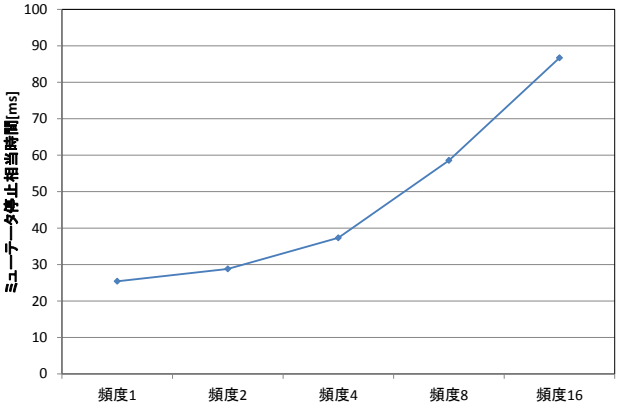

図 15. 書き換え頻度と停止相当時間

Vol.2014-GN-90 No.19
Vol.2014-CDS-9 No.19
Vol.2014-DCC-6 No.19
2014/1/24

3.7 Dirty カード数

本節では、Dirty カード数と書き換え頻度の関係を調査した結果を示す。3.5 節の実験で、リンク書き換え頻度が増加すると、リマークのみ増加することが確認された。これは、コンカレントマーク中に発生するオブジェクトのリンクの変化の整合性をリマークで取るためだと考えられる。Dalvik VM のコンカレント GC はカードテーブルを用いており、コンカレントマーク中の変更をカードテーブルのカードを Dirty にすることで記録する。リマークでは、このカードテーブルの Dirty カード部分のみ調査することで、マーク時間を削減している。

書き換え頻度の変化による Dirty カード数の変化を図 16 に示す。この図から、書き換え頻度増加に伴い Dirty カード数が増加することが確認でき、これが 3.5 節のリマーク時間の増加の原因と考えられる。

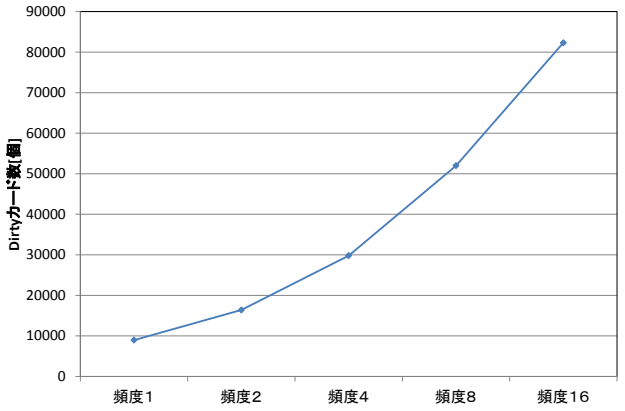

図 16. 書き換え頻度と Dirty カード数

4. STW 時間短縮の考察

前章では GC の基本測定を行った。測定より、オブジェクト数が多くかつ、オブジェクトに対する変更が多い場合を除き、STW 時間は 10ms 未満であり、オブジェクト数が多くかつ変更頻度が高い場合にのみ 50ms を越えるという結果が得られた。以上より、書き換え頻度が高い環境におけるリマーク処理による STW 時間の短縮により STW によるアプリケーション停止を大きく抑制できると考えられる。

リマーク処理による STW 時間の短縮についてはリマーク処理もミューテータとコンカレントに行い、リマーク中に生じた変更の整合性を取る STW 処理であるリリマーク処理を追加することにより実現できると考えることができる。同様にリリマークによる STW 時間が問題となる可能性が考えられるが、この問題はリリマーク処理のコンカレント化により改善ができると期待できる。一方、上記のようにリマークのコンカレント化などを追加で実装していくと、不要な処理の追加を招く可能性が考えられるが、図 11、図 16 の結果から、Dirty カード数と整合処理の時間強い相関があることが分かり、Dirty カード数の大小によりさらなるコンカレント処理の追加と STW 処理を動的に選択することによりこの問題を回避できると考えられる。

5. おわりに

本稿では, Dalvik VM のコンカレント GC の時間に着目し, 非ゴミオブジェクト数やリンク書き換え頻度の変更による GC の各フェーズの処理時間の測定を行った。また、Android の Dalvik VM 中にこの Dirty カード数をカウントする実装を施すことにより、STW 時間の短縮（ミューテータの性能低下抑制）に関する考察が可能となった。

今後の課題としては、考察した手法を端末に実装し、評価を行う。

謝辞

本研究は JSPS 科研費 24300034, 25280022 の助成を受けたものである。

参考文献

1) TOMOHARU UGAWA, HIDEYA IWASAKI, TAIICHI YUASA, “Improvements of Recovery from Marking Stack Overflow in Mark Sweep Garbage Collection”, IPSJ, Vol.5, No.1 (2012).
2) Patric Dubroy, “Memory Management for Android Apps”, Google I/O 2011