July 21, 2009
RT0868
Computer Science 6 pages

Research Report

Reducing GC pause time at Web application servers

Yohei Ueda, Hiroshi Inoue

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Webアプリケーションサーバでの GC停止時間の削減手法

上田 陽平　井上 拓
日本アイ・ビー・エム(株) 東京基礎研究所

Reducing GC Pause Time in Web Application Servers

Yohei Ueda, Hiroshi Inoue
IBM Tokyo Research Laboratory

1. はじめに

メモリ管理にガーベジ・コレクション(GC)を用いるJavaなどの言語処理系において，GCによるアプリケーション・プログラムの停止時間が，性能に大きな影響を与えることがある．特に，世代別GCにおけるFull GC (Major GC)は停止時間が数十秒から数分かかる場合があり，その性能への影響は無視できない．例えば，Javaで実装されたWebアプリケーションにおいて，トランザクションの最中にGCが発生してしまうと，そのトランザクションの応答時間を劣化させ，応答時間のQoS条件を満たせなくなったり，そのトランザクションのためにアプリケーションがデータベースのロックを確保した場合には，データベースを共有する他のアプリケーションが長時間DBのロック待ち状態になったりといった問題が生じる．

一般的なWebアプリケーションの実行環境としては，ひとつのWebアプリケーションを複数のJVMで稼動し，負荷分散装置でWebリクエストを転送するJVMクラスタ環境がよく用いられている．これは，複数のマシンを利用することにより性能向上，および，冗長構成による高可用性の実現が目的である．この様な環境であれば，長時間のFull GC停止時間が発生したJVMへのリクエストの転送を停止すれば，性能劣化を防ぐことができる．しかし，個々のJVMのGCは非同期で発生するため，タイミングによっては，複数のJVMがFull GCのために停止している場合が発生する可能性がある．その場合，Full GCを行っていないJVMへWebのリクエストの転送が集中してしまい，応答時間の劣化を招く．最悪の場合は，クラスタ内のすべてのJVMがFull GC中となり，この場合は新規リクエストをまったく処理できない状態となる．

本論文では，Webアプリケーションが稼動するJVMクラスタ環境において，Full GCが発生しているJVMの数を高々1個に抑え，Full GC中のJVMには新規Webリクエストの転送を停止することにより，長時間のFull GC停止時間によるアプリケーションへの影響を削減する技術について検討する．そのための構成手段は以下の通りである．

1. 負荷分散装置において，クラスタ内の各JVMのGCに関する情報（ヒープの残り空き容量，平均スループット，1トランザクションあたりの平均メモリ・アロケーション量，Full GCの停止時間など）を定期的に収集する．
2. 負荷分散装置は，1.で収集した情報をもとに，次にFull GCを起動するJVMを選択し，そのJVMへのWebリクエストの転送を停止する．
3. 負荷分散装置は，Webリクエストの転送を停止したJVMに対してFull GCを指示する．
4. Full GCが完了したJVMは，負荷分散装置にGC完了を通知する．
5. 負荷分散装置は，Full GCが完了したJVMへのWebリクエストの転送を再開する．

ただし，2.でFull GCを起動するJVMおよび選択するタイミングは次の通りである．

- ヒープ空き容量が最も少ないJVMを選択する．
- 選択したJVMのFull GC中に別のJVMのFull GCが開始しないように，空き容量が後述する計算式よる閾値以下になったタイミングでFull GCを起動する．

上記の手法により，Full GCによるWebアプリケーションの応答時間の劣化やDBの長時間ロックといった問題を解消することができる．

2. 関連研究

GCを用いるシステムにおけるGC停止時間の問題は広く知られており，例えばreal-time GC [1]や，concurrent GC [2, 3]， incremental GC[4]と言った手法が提案されている．これらの手法はFull GCの停止時間を削減するかわりに，アプリケーションのスループットが低下するといったトレードオフがある．本論文では，独立したトランザクションを繰り返し処理し，そのトランザクションの最中でなければGCの停止時間が大きな問題にならないというWebアプリケーションサーバの特性を利用した解決方法を検討する．

3. 負荷分散装置でのGCの監視

本論文では2種類のJVM内のメモリ消費量を計測することで，Webトランザクション処理中のGCを削減する手法を検討する．まず本節ではこの計測を負荷分散装置で行う場合について述べる

3.1 GC monitor

図1が本手法のシステム構成である. 負荷分散装置を拡張し, GC monitorというコンポーネントを追加する. 図中のDispatcherは実際にWebリクエストを各JVMへ転送するコンポーネントであるGC monitorは, 各JVMおよびDispatcherから情報を収集する機能, その情報を元にFull GCを開始するJVMの選択する機能, DispatcherへJVMへのリクエスト転送の停止・開始の機能, 各JVMへFull GCの開始の指示する機能などを持つ.

図2がGC monitorの挙動をあらわしたフローチャートである. GC monitorは定期的に各JVMの情報を収集し, それにもとづいてFull GCを起こすかどうか, Full GCを起こす場合にはどのJVMで起こすかを判断している. Step 1は, その定期的な情報収集を行うループの先頭であり, 一定時間の待機を行う. Full GCは数十分から数時間に一度発生するものであるので, 待機する時間は数十秒か数分程度である. 待機時間が完了したGC monitorはStep 2へ遷移し, 各JVMの情報を収集する. その情報をもとにStep 3でFull GCを起こすかどうかの判定を行う. Full GCを起こす場合にはStep 4へ遷移し, 最もヒープ空き容量の少ないJVMのFull GCを起動し, GC完了後にループの先頭のStep 1に戻り, 再び待機する. Step 3でFull GCを起こさないと判定した場合には, なにもせずにStep 1へ戻り, 再び待機する.

3.2 GC monitorが収集する情報

図2のフローチャートのStep 2において, GC monitorはJVMのヒープおよびスループットに関する情報を収集する. 図3がStep 2の動作のフローチャートであり, 収集する情報は下記の通りである. 変数の添字のkは, k番目のJVM (以降JVM$_k$と表す)の値であることを表しており, クラスタのn個のJVM について, 次の情報を収集する.

(1) 各JVMのGCに関する情報
- ヒープ容量 (H_k バイト)
- ヒープ空き容量 (F_k バイト)
- 前回のFull GCの時刻 (t_k)
- 前回のFull GCの停止時間 (T_k秒)

(2) Dispatcherの各JVMへのリクエスト転送に関する情報
- 平均スループット (X_k リクエスト/秒)

(3) リクエストあたりの必要メモリ量に関する情報
- 1リクエスト処理あたりのメモリ・アロケーション量の平均 (a_k バイト / リクエスト)

(1)のJVMのGC情報は, JMXなどの標準インターフェースを用いて取得することができる. また, JVMTIなどの拡張インターフェースを使用して, 独自の拡張モジュールをJVMに組み込んで収集することも可能である. Full GCの停止時間 (T_k)の変動に対応するために, 過去一定期間の停止時間を保存しておき, T_kとして過去一定期間の停止時間の平均を用いてもよい.

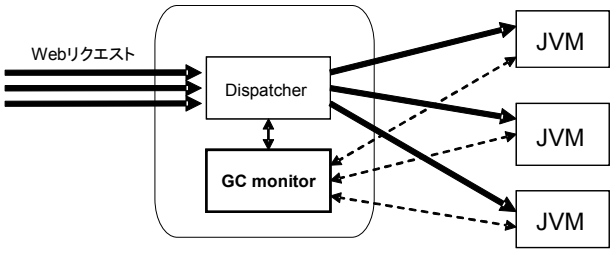

図 1. システム構成図

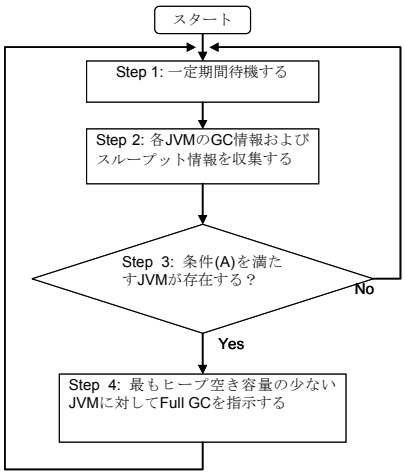

図 2. GC monitor の動作のフローチャート

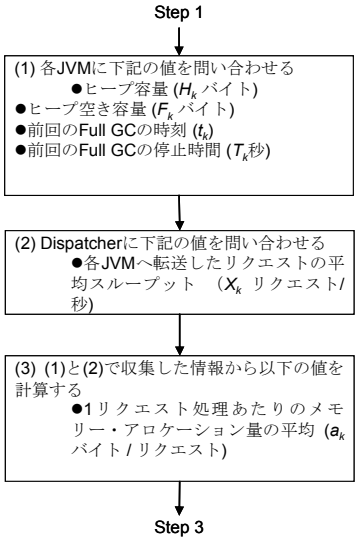

図 3. Step 2 の動作の詳細フローチャート

(2)のリクエスト処理に関する情報は, 通常Dispatcherは負荷分散のために各JVMへの転送量を管理しているので, その情報をGC monitorから参照可能にすればよい.

(3)の1リクエスト処理あたりのメモリ・アロケーション量の平均 (a_k)は, (1)および(2)の情報と現在の時刻(t_{now})から次のように計算できる.

$$a_k = (H_k - F_k) / \{ X_k \times (t_{now} - t_k) \}$$

リクエストあたりのメモリ・アロケーション量の変動に対応するために, 過去一定期間のリクエストあたりのメ

モリ・アロケーション量を保存しておき，a_kとして過去一定期間のリクエストあたりのメモリ・アロケーション量の平均を用いてもよい．

3.3 Full GCを行うJVMおよびタイミングの選択

Full GCを開始するJVMは，その時点でヒープ空き容量が最も少ないJVMである． ここで，説明を簡単にするために，ヒープ空き容量に関して $F_1 < F_2 < \cdots < F_n$ が成り立つと仮定する（これにより一般性を失うことは無い）．ヒープ空き容量が最も少ないのはJVM_1であるので，GC monitorは次にFull GCを開始するJVMとしてJVM_1を選択する．

通常，Full GCはヒープの空き容量が0になったタイミングで行う．しかし，Full GCを行っているJVMの数をクラスタ内で高々1個に抑えるためには，Full GCの開始をF_1が0になったタイミングで行うのでは遅すぎる場合が考えられる． つまり，JVM_2の空き容量F_2が少なくなっており，JVM_1のFull GCが完了する前に，JVM_2のヒープの空き容量が0になってしまう場合である．この場合，JVM_2でもFull GCを開始せざるをえず，Full GCを行っているJVMの数をクラスタ内で高々1個に抑えるという条件を満たすことができない．

この状況を防ぐためには，F_1が0になるより前に時間的余裕を持ってJVM_1のFull GCを開始し，JVM_2でFull GCが発生する前にJVM_1のFull GCを完了させればよい．JVM_1のFull GCの平均所要時間T_1秒経過後のJVM_2の予想空き容量は，JVM_2の平均スループット，1リクエスト処理あたりのメモリ・アロケーション量，および，現在のヒープ空き容量$_{F2}$から計算可能であり，$F_2 - a_2X'_2T_1$バイトとなる． ここで，X'_2は，JVM_1がFull GCを行っている時の，JVM_2の予想されるスループットである．JVM_1のFull GC中は，DispatcherはJVM$_1$へのリクエスト転送を停止し，残りのJVMへの転送をそれぞれの処理能力に合わせて比例して増やすため，$X'_2 = X_2 + X_1 \times X_2 / \Sigma_{j\neq 1} X_j$ となる．このT_1秒後のJVM_2の予想空き容量が0になることが判明したタイミングでJVM_1のFull GCを開始すればよい． 実際には，予想空き容量が0になるタイミングでは，平均よりスループットが増加していた場合などではFull GCが予想より早く必要になってしまうため，ある程度の余裕を持たせる必要がある． そこで，空き容量があらかじめ決めておいた閾値 α を下回ったタイミングでFull GCを開始する．つまり，定期的に空き容量を監視し，$F_2 < a_2X'_2T_1 + \alpha$ が成り立ったときにFull GCを開始する．

JVMが3個の場合(図4)も同様に考えられる．JVM_2に関して$F_2 < a_2X'_2 T_1 + \alpha$ が成り立つ場合に加えて，JVM_3に関して$F_3 < a_3X'_3 T_1 + a_3X''_3 T_2 + \alpha$ が成り立つ場合にもFull GCを開始する必要がある．ここで，X''_3 はJVM_2がFull GC中の場合のJVM_3の予想されるスループットであり，$X''_3 = X_3 + X_2 \times X_3 / \Sigma_{j\neq 2} X_j$ となる．$a_2X'_2 T_1$はJVM_1がFull GC中にJVM_3でアロケートされるメモリ量を表し，$a_3X''_3 T_2$はJVM_2がFull GC中にJVM_3でアロケートされるメモリ量を表している．

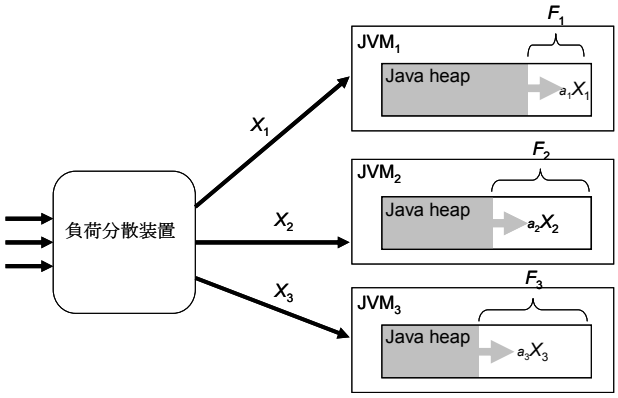

図 4.　JVM が 3 個の場合の Java ヒープの状態

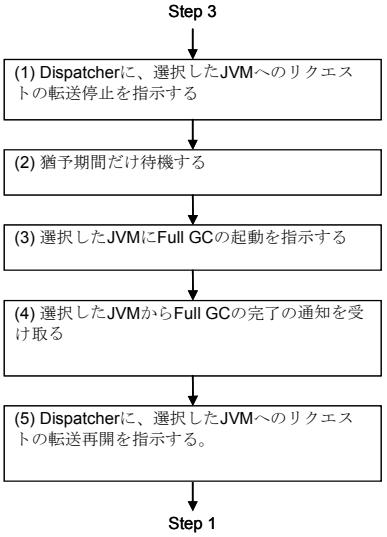

図 5.　Step 4 の動作の詳細フローチャート

以上の議論は，次のようにJVMがn個の場合に一般化できる．

あるk $(1 \leq k < n)$で

$$F_{k+1} < \Sigma_{i\leq k}\, a_{k+1} X^{(i)}{}_{k+1} T_i + \alpha \cdots \quad \text{(A)}$$

が成り立ったとき，JVM_1でFull GCを開始する．ただし，

$$X^{(i)}{}_{k+1} = X_{k+1} + X_i \times X_{k+1} / \Sigma_{j\neq i} X_j$$

である．

図2のフローチャートにあるように，GC monitorは定期的に，上記の条件(A)が合致するJVMが存在するかを確認している．条件(A)を満たすJVMが存在する場合には，Step 4へ遷移し，最もヒープ空き容量の少ないJVM_1のFull GCを開始する．条件(A)を満たすJVMが存在しない場合には何もせずにStep 1にもどる．

3.4 Full GCの実施

GC monitorが選択したJVMのFull GCを行う手順は図5の通りである．

(1)でリクエストの転送を中止してから，(3)でFull GC

を起動するまでの間に数秒の猶予期間(2)を挿入することにより，仕掛かり中のリクエスト処理がほぼ完了した後に，Full GCを行うことができる．これにより，仕掛かり中のリクエスト処理完了前にFull GCが発生し，応答時間が長くなることを防止できる． ただし，この方法だとリクエスト処理完了を完全に待っているわけではなく，なんらかの原因でリクエスト処理に長時間がかかっている場合などは，リクエスト処理中のFull GCを完全に防ぐことはできない．

Dispatcherがリクエスト処理の完了を監視することで，リクエスト処理中のFull GCを防ぐこともできる． ただし，ひとつでも完了しないリクエスト処理が残っているとFull GCが開始できないため，この場合でもタイムアウトは必要である．

(3) での Full GC の起動は，JVMTI の ForceGarbageCollection APIを使用することで実現可能である．(4)のFull GCの完了の通知もJVMTIのGarbage Collection Finishイベントを使用することで取得することが可能である．

4. Webアプリケーションサーバ でのGCの監視

本節ではトランザクションごとのメモリ消費量の計測をアプリケーションサーバ自身で行う場合について述べる．本手法は：

1. 1つのトランザクションあたりに新しくアロケーションするメモリ量に関するプロファイルを取得し，
2. トランザクションの実行開始前に必要量のメモリを事前に割り当て，
3. この事前割り当てに失敗した場合には，このアプリケーションサーバではこれ以上の処理を開始せず，アプリケーションサーバ内で処理中のトランザクションがなくなるのを待ち，GCを実行すること

を特徴とする．これにより，アプリケーションサーバがトランザクションを実行している最中にGCを実行すること防ぎ，かわりにDBロックが解放されていると考えられるトランザクションの境界においてGCを実行することで上記の問題点を解決することができる．

4.1 プロファイルの取得

プロファイルの取得のためには，トランザクションの処理中に新たにアロケーションするメモリの総量の計測を行う（図6）．実際に全てのメモリのアロケーションのログを取り正確なメモリのアロケーション量を調べることは可能だが，計測のオーバーヘッドが非常に大きくなる．本手法の目的では正確なバイト数ではなく概算でかまわないためよりオーバーヘッドが小さい手法を用いることができる．

多くのJava処理系では，マルチスレッドプログラムでのメモリのアロケーションを高速化するために，各スレッドにある程度大きなメモリブロック(例えば100KB単位など)を予め割り当て，各スレッドはそこから新しいオブジェクトの割り当てを行う実装になっている．このメモリブロックを使い切ると，新しいメモリブロックをランタイムシステム内のメモリアロケータに要求する．

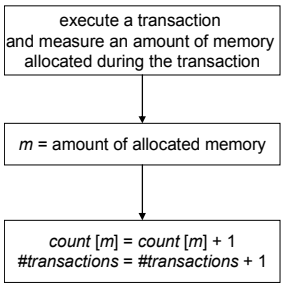

図 6. プロファイルの取得方法

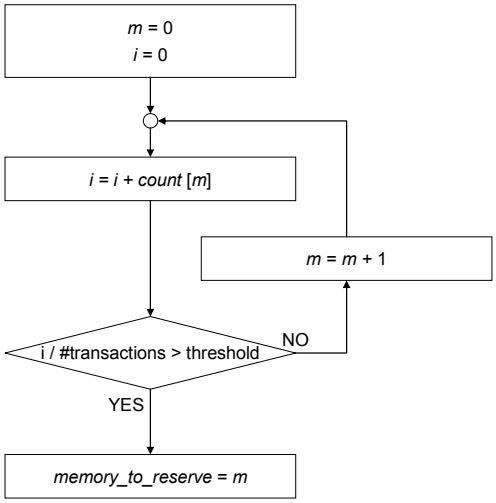

図 7. メモリの事前割り当て量の決定方法

一般にWebアプリケーションサーバでは，1つのWebトランザクションに対して1つのワーカースレッドが割り当てられ，そのスレッドがトランザクションの最後まで実行を行う．そのため，あるワーカースレッドがWebトランザクションの開始から終了までの間に，新しいメモリブロックをいくつ新たに割り当てたかを数えることで，低いオーバーヘッドでメモリ・アロケーション量の概算値を計測することができる．

計測されたプロファイル結果を統計処理することで，トランザクションの処理を開始する前に事前に割り当てるべきメモリ量を決定する．この時，単純に過去に計測された最悪値を用いてしまうと各トランザクションに対して極端に過大なメモリの割り当てを行ってしまう恐れがある．これは，各トランザクションで新たにアロケーションされるメモリは一般的に入力内容によって変動し，最悪値は例えばトランザクションの中で例外が発生したなどの場合，極端に大きくなる可能性がある．そのため，この最悪値に対応するメモリを常に事前に割り当てるのは効率が悪い．そこで，大半（例えば95%）のトランザクションが終了までに必要とするメモリ量を割り当てるようにする．このような場合には，トランザクションの途中で事前に割り当てられたメモリを

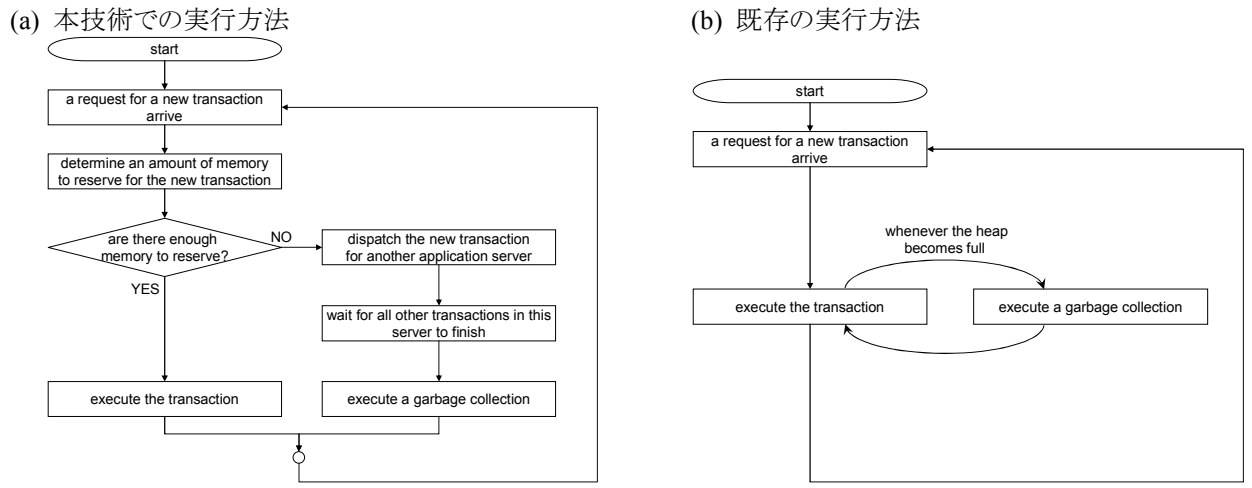

図 8. トランザクション処理の実行方法

使い切り，DBロック保持中にGCが発生してしまう場合がある．そのため，この閾値はメモリの使用量とDBロック保持中のGCが起こることのペナルティを考慮し決定する．このメモリの割り当て量を決定するためにはプロファイル時には単に平均量のみや，最悪値のみのプロファイル取得ではなく，トランザクションごとのメモリ使用量のヒストグラムを作成する．ヒストグラムの作成は単純なプロファイルの取得と比較するとメモリの使用量は多くなるが，前に述べたようにプロファイルの粒度は荒くてよいため，大きな問題にはならない．また，トランザクションの種類毎にそれぞれプロファイルを取得することで精度を向上することが可能である．

4.2 Webトランザクションの実行について

Webトランザクションを開始する場合には，まずプロファイル結果から必要なメモリの事前割り当て量を決定する（図7）．もし，ここでメモリの事前割り当てに失敗した場合には，このトランザクションの処理は開始せず，可能であればクラスタ内の他のアプリケーションサーバにディスパッチしなおすとともに，これ以降のリクエストの受付を停止する．トランザクションを開始しなかった場合にも，このアプリケーションサーバではすでに実行中の他のトランザクションが存在するため，これらの終了を待つ．全ての既存トランザクションの処理が終了したところでGCを起動する．GC処理が終了次第，新規トランザクション処理の受付を再開する．図8にこの処理のフローチャートを示す．

本手法で事前に必要と見込まれるメモリの確保を行っておくことで，ほとんどの場合にトランザクション処理の最中にメモリが不足することを防ぐことが可能になる．しかし，たとえ最悪値を考慮して事前に割り当てを行ったとしても，例えば特殊な例外が発生した場合など，トランザクションの実行中にメモリ不足による割り当ての失敗が起こる可能性は残る．そこで，あるトランザクションが事前に割り当てられたメモリを使い切った場合には，そのトランザクションの処理をいったん停止し，他の全てのトランザクションが終了もしくは同様にメモリ割り当ての失敗で停止した状態になるまで待つ．その後，停止したトランザクションの処理を再開する．他の終了したトランザクションが事前に割り当てを受けたメモリを使い切らずに終了した場合には，そのメモリを使用してGCを起こさずにこのトランザクションを終了できる可能性がある．ここでも再びメモリの割り当てに失敗した場合にはGCを実行する．この時点ではこのアプリケーションサーバ内の大半のトランザクションは既に終了しているため，他のトランザクションの終了をまたずにGCを起こすよりもこのアプリケーションサーバがDBロックを取得している可能性をはるかに小さくすることができる．

5. まとめ

本論文では，Web アプリケーションが稼動する JVM クラスタ環境において，Full GC の停止時間がアプリケーションの応答時間にあたえる影響を削減する技術を提案した．さらに，トランザクションごとのメモリ消費量の計測をアプリケーションサーバ自身で行い，DB ロック保持中の Full GC を回避する手法も提案した．これらの手法により，Java アプリケーションサーバを使用した Web アプリケーションを運用する際にしばしば発生していた Full GC による性能劣化を防ぐことができ，よりスケーラブルで長時間連続運用に耐えられる Web システムの構築が可能となる．

参考文献

[1] David F. Bacon, Perry Cheng, and V.T. Rajan, “A Real-time Garbage Collector with Low Overhead and Consistent Utilization”. in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 285-298, 2003.

[2] Henry G. Baker. “List processing in real-time on a serial computer”. *Communications of the ACM* 21(4):280.94, 1978.

[3] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. “Mostly parallel garbage collection”. in

Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, pp. 157-164, 1991.

[4] Tony Printezis and David Detlefs. “A generational mostly-concurrent garbage collector”. in *Proceedings of the ACM SIGPLAN international symposium on memory management,* 2000.