システムソフトウェアと
オペレーティング・システム 92−18
（2003. 2. 28）

組み込みシステム向け JVM における実時間 GC のスケジューリング

竹内 悟† 平沼義直† 井上泰彰†

Java 仮想機械の GC を実時間化し，GC によるプログラムの中断時間を減少させる試みがなされている．しかし，PC 等の高性能な機器と比較し処理能力が格段に劣る組み込みシステム上において，実時間 GC はプログラム全体のパフォーマンスを著しく低下させてしまう．

本論文は，実時間性を保ちつつ，パフォーマンスを維持するため，プログラムが許容する GC による中断時間を基準とし，可能な限り GC 作業を連続して行うスケジューリング方法を提案する．我々のスケジューリング方法を，組み込みシステム向け JVM 上に実装し，評価したところ最大 70 % 程のパフォーマンス向上が見られた．

Scheduling Real-Time Garbage Collection on Java Virtual Machine for Embedded System

Satoru Takeuchi,† Yoshinao Hiranuma† and Yasuaki Inoue †

Recently, Real-Time GC is implemented on Java Virtual Machine (JVM) to gain high responsiveness by reducing GC intervals. Real-Time GC, however, can lead to performance degradation.

This paper proposes a novel algorithm, which is based on real-time demand, for scheduling the work of GC. We implemented this algorithm on JVM for embedded system, and the maximum improvement was 70 %.

1. はじめに

近年，Java 仮想機械 (以下，JVM) は一般的な PC のみならず，携帯電話，デジタルテレビなどの組み込み機器にも広く導入されるようになった．JVM の特徴の一つに，**ごみ集め (以下，GC)** による自動メモリ管理がある．通常 GC は，開始から終了まで，実行中のスレッドを中断して行われる．この中断時間は，実行するプログラムの規模によっては，数ミリ秒から数秒となり，対話型アプリケーションなどに多大な悪影響を与えてしまう．そこで，作業を分割して GC 以外のスレッドと交互に行う，**実時間 GC** が考案された．現在，より厳しい実時間性を要求する組み込みシステム向け JVM に，実時間 GC を適用する動きが活発である[1)2)]．

しかし，実時間 GC は，作業を一括で処理する GC と比較し，GC の開始から終了までの時間が長くなり，プログラム全体のパフォーマンス悪化につながる．よって，実時間性を保ちつつパフォーマンスを維持するため，GC を効率良くスケジューリングする必要がある．

本論文は，2 章にて実時間 GC の問題点を述べる．3 章は本論文の主題であり，問題点を解決するための実時間 GC のスケジューリング方法を提案する．4 章ではそのスケジューリング方法を組み込みシステム向け JVM へ実装する．5 章にて提案手法の評価実験を行う．6 章で関連研究を紹介し，7 章で結論と今後の課題を述べる．

2. 実時間 GC の問題点

本章は，最初に GC の作業内容を説明し，それを踏まえた上で実時間 GC の問題点を述べる．

GC は，各スレッドによりオブジェクトが動的に割り当てられる領域 (以下，ヒープ領域) 中の，「どのオブジェクトからも参照されないオブジェクト」 (ごみ) を自動的に回収し，再利用する．GC は，開始されると，各スレッドがいつでも参照可能な領域 (以下，ルート領域) から辿れるオブジェクトに印をつけていく．辿れるオブジェクト全てに対して作業が終了した時点で，印がついていないオブジェクトを回収する．このよう

† 三洋電機株式会社 デジタルシステム研究所
SANYO Electric Co., Ltd. Digital Systems Development Center

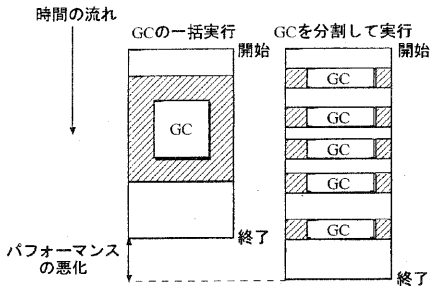

図 1 GC の分割によるパフォーマンスの悪化

に, GC の作業対象となる領域はルート領域とヒープ領域に分けられる.

さて, GC を分割して実行している間, ヒープ領域は GC と GC 以外のスレッド, 両者に操作される. よって, ヒープ領域の無制限な変更を防ぐため, GC 以外のスレッドによるヒープ領域の操作に, なんらかのチェック作業が必要となる. また, ルート領域は, GC 以外のスレッドにより随時更新されるため, GC のアルゴリズムによっては何度もルート領域から辿りなおす必要がある. つまり, GC を分割すると,

- 実行中プログラムの負荷の増加
- GC 作業量の増加

が生じ, 図 1 に示すようにプログラム全体のパフォーマンスが悪化してしまう. また, 実時間 GC を実装した JVM を, PC 等の高性能な機器と比較し処理能力が格段に劣る組み込みシステム上で動作させると, その影響はより大きくなると考えられる.

3. 実時間 GC のスケジューリング

スレッドには, 実時間性の (要求が) 高いものと低いものが存在し, 常に実時間性の高いスレッドに合わせて GC を分割するのは, 前章で述べたようにパフォーマンスの観点から見ると無駄である. 本章は, 動作しているスレッドの実時間性に応じて GC の分割具合を調整し, 実時間性を保持しつつ最大限 GC を行いパフォーマンスを維持するスケジューリング方法を提案する.

3.1 GC による中断時間の定義

まず, 準備として GC による中断時間の明確な定義付けを行う. 以降は, 断わらない限り対象とするシステムが,

- ヒープ領域の操作は, ヒープに関するロック (以下, ヒープロック) を取得したスレッドのみ行える.
- GC は, オブジェクト割り当て時にヒープ領域の空きサイズが, ある閾値を下回った時 (以下, GC 必要時), 割り当てを試みたスレッドが現在の作業を中断して行う.

といった特徴を持つとする. このようなシステムでの通常時のオブジェクト割り当ては, 図 2 左側に示すように,

(I) ヒープロックを取得し,
(II) ヒープ領域の一部にオブジェクトを割り当て,
(III) ヒープロックを解放する.

という流れで行われる. しかし, GC 必要時は図 2 右側に示すように, 割り当ての前に GC を行う必要がある. このような場合の, GC による中断時間を,

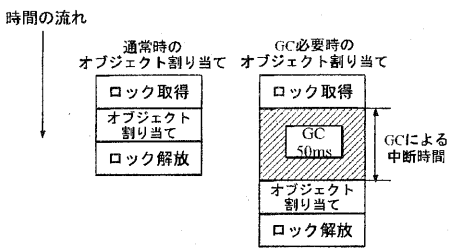

図 2 GC による中断時間

GC による中断時間
オブジェクト割り当て時の GC 作業時間

と定義する.

3.2 許容中断時間の導入

提案するスケジューリング方法は, GC を開始したスレッドの実時間性に応じて GC による中断時間を調整する. そのため, 各スレッドに実時間性の具体的な指標が必要となる. その指標として, 以下で定義する許容中断時間を付加する.

許容中断時間
スレッドが許容できる, GC よる中断時間の最大値

逆に言うと, この許容中断時間が満たされている限り, システムの実時間性は保持されている.

3.3 許容中断時間に基づいたスケジューリング方法

前節で導入した許容中断時間を使用し,

スケジューリング方法
GC をスレッドの許容中断時間分行う

を提案する. 図 3 は上記の方法を使用した GC 必要時のオブジェクト割り当て処理である. 図 3 に示すように, 各スレッドはそれぞれの許容中断時間分だけ GC を行う. このスケジューリング方法により, GC

をスレッドが許容する限り最大限行うことが可能となり，パフォーマンスは維持される．さらには，許容中断時間を保証するため，システムの実時間性も保持できる．

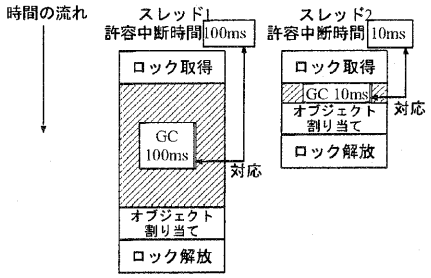

図 3 スケジューリング方法を用いた GC 必要時のオブジェクト割り当て処理

3.4 許容中断時間に基づいたスケジューリング方法の実現

本節は，前節のスケジューリング方法を，複数のスレッドがそれぞれ異なる許容中断時間を持ち，同時に動作する JVM 上で実現する方法に関して述べる．

複数のスレッドが同時に動作する環境では，許容中断時間の長いスレッドによる GC 中に，許容中断時間の短いスレッドによる，オブジェクト割り当て処理が発生し得る．そのような場合は，後者の許容中断時間に基づいて，GC を停止する必要がある．その目的のため，各スレッドの**残存中断時間** (許容中断時間 − その時点までの GC による中断時間) をリストとして保持し，残存中断時間が 0 になったら処理を停止する GC (**以下，プリエンプティブ GC**) を設計した．このプリエンプティブ GC を用い，許容中断時間 300ms のスレッド 1，50ms のスレッド 2，そして 20ms のスレッド 3 による作業の流れと，残存中断時間のリストの状態遷移を図 4 に示す．以下は，図 4 中の番号に対応した各スレッドの処理の解説である．

(i) スレッド 1 は残存中断時間 300ms を挿入した後，ヒープロックを取得し，リスト中の残存中断時間の最小値 300ms を用いて，GC を開始する．

(ii) スレッド 2 は残存中断時間 50ms を挿入した後，ヒープロック待ち状態に移行する．

(iii) スレッド 1 はリストの状態を更新し，リスト中の残存中断時間の最小値 50ms を用いて GC を再開する．

(iv) スレッド 3 は残存中断時間 20ms を挿入した後，ヒープロック待ち状態に移行する．

表 1 プリエンプティブ GC を用いた GC 必要時のオブジェクト割り当てアルゴリズム

```
1   begin
2   リストに残存中断時間を挿入する;
3   リスト中の残存中断時間を更新する;
4   ヒープロックを取得する;
5   while リスト中の残存中断時間の最小値 != 0 do
6     GC を全スレッドの許容中断時間の最小値分行う;
7     リスト中の残存中断時間を更新する;
8   オブジェクト割り当てを行う;
9   リストから残存中断時間を取り外す;
10  ヒープロックを解放する;
11  end;
```

(v) スレッド 1 はリストの状態を更新し，リスト中の残存中断時間の最小値 20ms を用いて GC を再開する．

(vi) スレッド 1 はリスト中の残存中断時間の最小値が 0 となったため，GC を停止し，オブジェクト割り当てを行い，自身をリストから取り外しヒープロックを解放する．

(vii) スレッド 3 はヒープロックを取得し，リスト中の残存中断時間の最小値が 0 であるため，GC を行わず，即座にオブジェクト割り当てを行い，自身をリストから取り外しヒープロックを解放する．

(viii) スレッド 2 はヒープロックを取得し，リスト中の残存中断時間の最小値 10ms を用いて，GC を開始する．

(ix) スレッド 2 はリスト中の残存中断時間の最小値が 0 となったため，オブジェクト割り当てを行い，自身をリストから取り外しヒープロックを解放する．

また，表 1 は プリエンプティブ GC を用いた GC 必要時のオブジェクト割り当てのアルゴリズムである☆．表 1 の 6 行目において，全スレッドの許容中断時間の最小値毎にリストを更新する．これにより，全てのスレッドの許容中断時間に基づいて GC を停止できる．このプリエンプティブ GC を用いれば，提案したスケジューリング方法を JVM 上で実現できる．

3.5 許容中断時間に基づいた動的なスケジューリング方法

前節までで提案したスケジューリング方法は，プログラム開始時に許容中断時間を設定し，それに基づいて行う静的な方法である．静的なスケジューリング方法では，許容中断時間が短かく，割り当てるオブジェク

☆ 実際には，リストに残存中断時間を挿入する段階でヒープロックを持つスレッド (GC を行っていたスレッド) は，挿入を行ったスレッドの優先度を継承する (priority inheritance)．

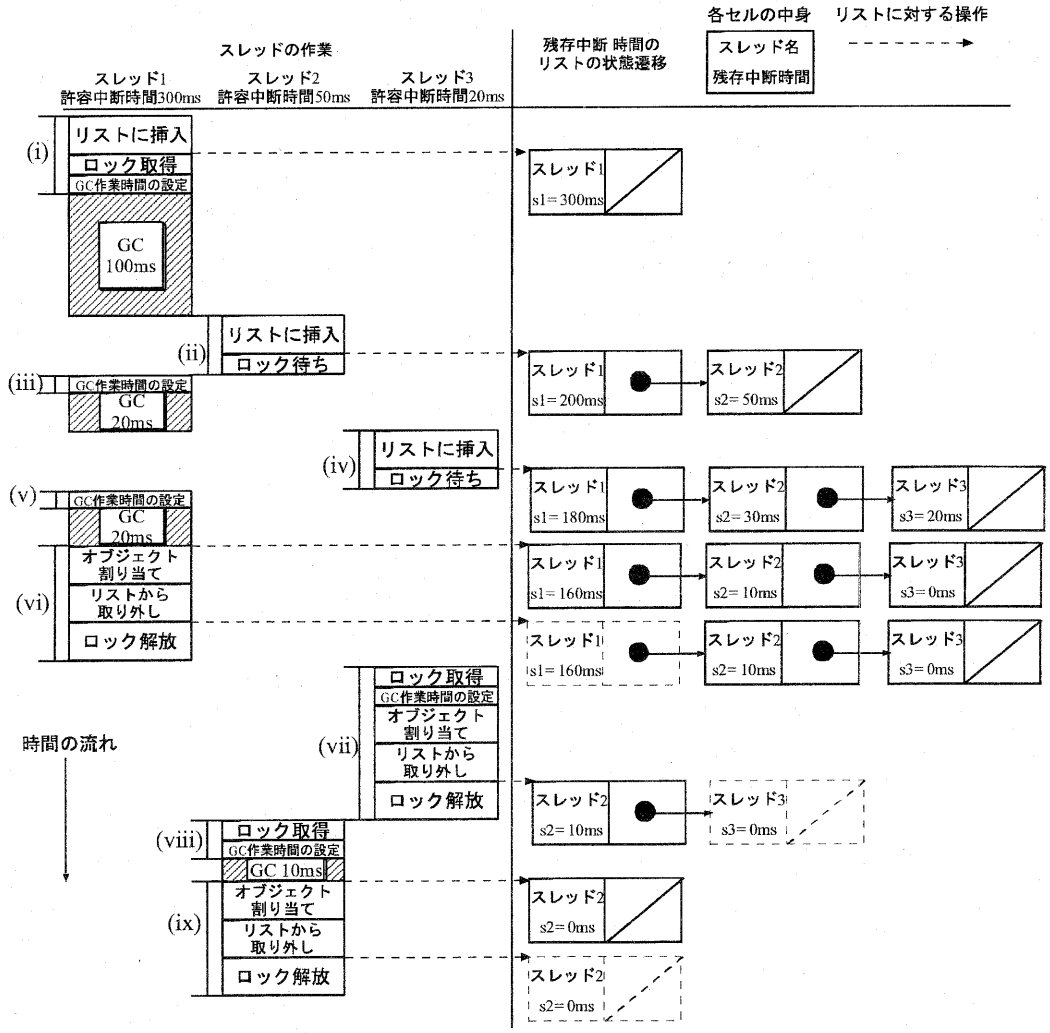

図 4　スレッドの作業フローと残存中断時間のリストの状態遷移

トのサイズが大きいスレッドが存在する場合, GC 終了前にオブジェクトの割り当てが行えなくなる飢餓状態 (starvation) を引き起こす可能性がある. 飢餓状態が生じると, PC 等のリソースが豊富な機器では, ヒープ領域を拡張し対処する. しかし, リソースの限られた組み込み機器では, そのような対処法は現実的でない. よって, 飢餓状態が生じた時点で GC を最後まで行なう必要があり, スレッドの許容中断時間を大幅に越えてしまう. 飢餓状態を回避するためには, GC 中のヒープ領域の空きサイズや, 単位時間に割り当てられたオブジェクトのサイズなどに応じて, スレッドの許容中断時間を動的に変更する必要がある. 例えば, 各スレッドに通常時の許容中断時間とは別に, 最悪許容中断時間を用意しておき, 飢餓状態に陥いる危険性がある場合は, 通常時の許容中断時間から最悪許容中断時間に変更して再度スケジューリングを行う. このような動的なスケジューリング方法を用いれば, システムの実時間性を最大限保持できる.

4. 実　　装

本章にて, 前章で提案したスケジューリング方法を組み込みシステム向け JVM 上へ実装する. 4.1 節ではスケジューリング方法の適用先である実時間 GC の設計, 4.2 節では実装環境に関して述べる.

4.1 複写方式実時間 GC アルゴリズム

今回我々は, 複写方式 GC (Copying GC) の実時間化を行った. 複写方式 GC は, ヒープ領域を From 空間 と To 空間に分け, ルート領域から辿れるオブジェクトを To 空間へ複写し, 印付けと回収を同時に行う手法である. 複写方式 GC を実時間化するため, 2 章で述べた, GC 中のヒープ領域に対する操作へのチェック作業を, 書き込み時に行う方式 (以下, ライト・バリア) を用いたアルゴリズムを設計した. 本アルゴリズムは, Brooks が提案したアルゴリズム[3] を応用したものである. 以下, GC 開始から終了までの流れである. また, 図 5 は各 Step の作業内容を表している.

Step 1. GC 開始時の作業

ルート領域から直接参照されているオブジェクトを To 空間へ複写する. 複写先のアドレスはオブジェクトと別領域に保存し, オブジェクト自身はそのまま保持する. また, ルート領域中のポインタも From 空間を指したままにしておく.

Step 2. GC の漸時実行

To 空間内に From 空間を指すポインタがなくなるまで, 複写されたオブジェクトを走査する. ポインタが From 空間を指していた場合, 以下の作業が発生する.

(1) 参照先のオブジェクトが複写されていない ⇒ オブジェクトを To 空間に複写し, ポインタを複写先のアドレスに変更する. また, 複写元オブジェクトはそのまま残しておく,

(2) 参照先のオブジェクトが複写済み (別領域に複写先のアドレスが書き込まれている) ⇒ ポインタを別領域に書き込まれている, 複写先のアドレスに変更する.

また, この間, GC 以外のスレッドによるオブジェクト割り当ては, To 空間の末尾に行われ, オブジェクトに対する書き込みは, ライト・バリアによってチェックされる.

Step 3. GC 終了時の作業

ルート領域中のポインタの参照先を, 別領域に書き込まれているオブジェクトの複写先アドレスに変更する. From 空間と To 空間を交換し, GC を終了する.

4.2 実装環境

前節で実時間化した GC に提案したスケジューリング方法を適用したものを, 表 2 に示す環境上に実装した. ここで, C Virtual Machine (以下, CVM)[4] は SUN Microsystems より配布されている, 組み込みシステム向けの JVM である.

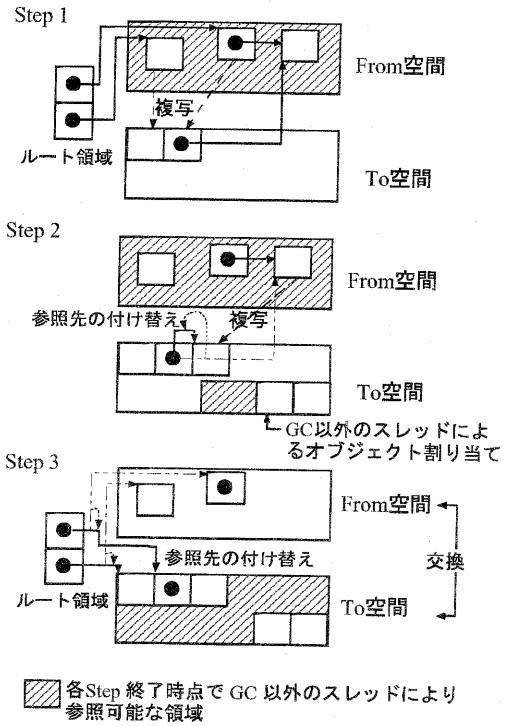

図 5 複写方式実時間 GC

表 2 実装環境

機器種	デジタルカメラ
CPU	RISC プロセッサ 28.7MHz
RAM	16MB
実時間 OS	μ ITRON 3.0 準拠
JVM	C Virtual Machine

5. 評価実験

前章で構築した環境上で, 提案する手法を以下の 2 項目に関して評価する.

パフォーマンス

同程度の実時間性を持つ作業時間が均一な実時間 GC (以下, 単純な実時間 GC) と比較した, パフォーマンス

実時間性 (GC による中断時間)

作業を一括で行う GC (以下, 一括型 GC) と比較した, 割り込みにより起動する許容中断時間の短いスレッドの GC による中断時間

評価には, 以下の 3 種類のプログラムを用いる.

SimpleObject Benchmark Program(Simple)

単純なオブジェクト生成プログラム. 10 バイト

の行列を, 30 万個生成する.

Minibenchmark Program[5](Mini)

実行するスレッドの数, 仕事の数, 仕事量の最小値と仕事量の最大値を設定し実行する, マルチスレッドベンチマークプログラム. 仕事の内容は, 文字列操作である. 本実験では, スレッド数を 3, 仕事数を 3, 仕事の最小値を 10, そして最大値を 100 とした.

Jini Benchmark Program(Jini)

Jini Network Technology[6] を用い, ネットワーク上に存在するサービスをルックアップサーバを用い検索し, 発見したサービスを使用するプログラム. 本実験では, ルックアップサーバを 1 つと, デジタルカメラ中の写真を画面上に表示するディスプレイサービスを 1 つ用いた.

提案する GC では, 上記 3 種類の評価プログラムを実行するスレッドと, 割り込みにより起動するスレッドにそれぞれ, 表 3 の許容中断時間を設定した. 表 3 中の単位オブジェクト走査時間とは, 1 つのオブジェクト中のポインタを全て走査し, それぞれの参照先オブジェクトを複写するのに要する時間である.

表 3 許容中断時間の設定

	許容中断時間
割り込みにより起動するスレッド	単位オブジェクト走査時間
評価用プログラム実行スレッド	全 GC 作業時間の 1/3

また, 比較対象の一括型 GC として CVM に備わっている複写方式 GC を, 単純な実時間 GC として, 4 章で設計した複写方式実時間 GC の作業時間を単位オブジェクト走査時間に統一して実行するものを用いる.

次節以降の実験は,

- GC は From 空間の空き領域が 20 %以下となった時に開始される
- 1.5 MB のヒープ領域を 750KB ずつに分け From 空間, To 空間に割り当てる

といった条件の元で行われた. また, 各評価用プログラムは 3 回ずつ実行し, 評価にはその平均値を用いている.

5.1 パフォーマンスの向上

まず, 提案する GC と単純な実時間 GC のパフォーマンスを比較した. 図 6 は, GC の総作業時間を表わしたグラフである. グラフより Simple, Jini に関しては 35 % 程, Minibenchmark に関しては 70 % ものパフォーマンス向上が見られる. また, 図 7 のグラフから単純な実時間 GC と比較して GC の起動回数が

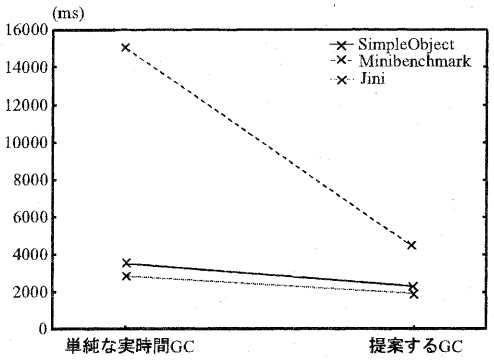

図 6 GC の総作業時間

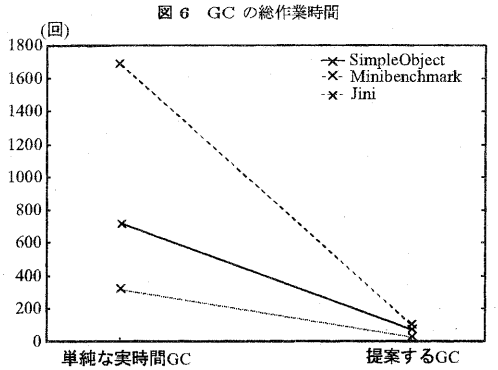

図 7 GC 起動回数

表 4 ライト・バリアへのヒット回数 (回)

	Simple	Mini	Jini
単純な実時間 GC	0	2546	1792
提案する GC	0	**340**	**219**

大幅に少ないこともわかる. 表 4 は, GC 中のライト・バリアへのヒット回数を表わしている. GC 中の時間が短かくなり, ヒット回数が減少していることがわかる. これらの結果より, 提案する GC によるパフォーマンスの向上を確認できた.

5.2 GC による中断時間の短縮

次に, 提案する GC と一括型 GC の実時間性を比較した. 具体的には, 評価用プログラムを実行しておき, 1 秒間に 2 回ランダムに割り込みを与えスレッドを起動した. 割り込みにより起動したスレッドの, GC による中断時間の平均と, その最大値を, グラフ化したのが図 8 である. 各グラフから, 一括型 GC と比較し, 最大中断時間が 60 % から 80 % 程減少したことがわかる. また, 平均中断時間は, Simple を実行した際 90 % 程減少している. これらの結果より, 実時間性の大幅な向上が確認できた.

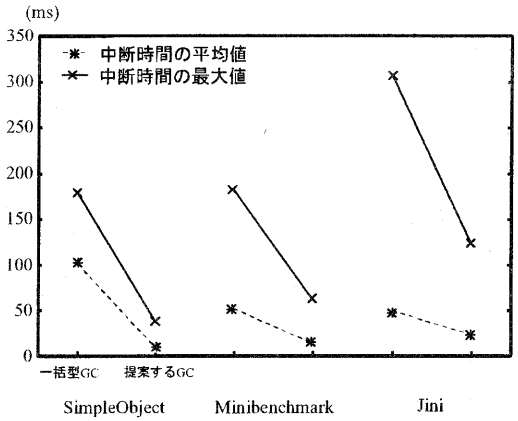

図 8 割り込みにより起動したスレッドの GC による中断時間

6. 関連研究

組み込みシステム向け JVM における, GC の実時間化として, GC を一つの独立したスレッドとして起動し, 他のスレッドと並列して行う, 並列 GC (Concurrent GC) がある. 栗林[1] は, GC スレッドの優先度を, 一定周期で切り替える手法を提案した. また, 奥村他[2] は, GC を必要とするスレッドの優先度に基づいて, GC スレッドの優先度と, 作業時間を定める手法を提案した. 本手法は, これらの手法と比較し GC をスレッドとして起動する必要がなく, GC 起動時にコンテキストスイッチを伴わないため, 負荷が軽くなると考えられる.

Java Community Process (JCP)[7] プログラムの Java Specification Requests (JSR) にて, 公開されている Real-time Specification for Java[8] は, Java プログラムを実時間システム上で実行する際, 必要とされるクラスを標準化した仕様である. 仕様中で定義されたクラスは, 実行周期と, その周期に対する遅延のデッドラインをパラメータとして持つ. 許容中断時間は, そのデッドラインに基づいて自動的に設定可能である.

また, Richard 他[9] は, 複写方式 GC の実時間化を行い, JVM における GC によるプログラム中断時間の短縮を行った. しかし, GC 中のオブジェクト割り当てが GC 対象とならない領域に行われるため, ヒープ領域の拡張が必要となり, リソース制限の厳しい組み込みシステム向きではない.

7. まとめと今後の課題

本論文は, 実時間 GC を効率良くスケジューリングし, 組み込みシステム向け JVM において, 実時間性とパフォーマンスを両立させる手法を提案した. 実験の結果, 同程度の実時間性を持つ GC と比較し, 最大 70 % ものパフォーマンス向上が見られた. 今後の課題として,

作業時間に上限がつけられる実時間 GC

4 章で提示したアルゴリズムは, GC 1 回あたりの作業時間に上限がつけられない. 例えば Step1 は一括で行う必要があり, ルート領域の大きさによっては, GC による中断時間が大きくなってしまう. 許容中断時間を確実に保証するため, 作業時間に上限をつける工夫[9] が必要となる.

を実現し, 本スケジューリング方法の精度を, さらに増すよう取り組む予定である.

参 考 文 献

1) 栗林 博. Java™ のリアルタイム拡張の動向. **電子情報通信学会「コンピュータシステム」実時処理に関するワークショップ** *(RTP'99)*, 1999.
2) Gen Okuyama, Eiji Takimoto, Masahito Shiba, and Eiji Okubo. Java Virtual Machine on Easel Real-Time Operating System. Technical Report 89, IPSJ SIGNotes Contents system software and Operating System, February 2002.
3) Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. *In Conference Recod of the 1984 ACM Symposium on LISP and Functional Programming*, 1984.
4) SunMicrosystems Inc. Connected Device Configuration(CDC) and the Foundation Profile.
5) `http://www.cse.psu.edu/~gchen/kvmgc/`.
6) Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini[tm] Specification.* Addison Wesley, 1999.
7) The JavaCommunity Process. `http://jcp.org/`.
8) The Real-Time Specification for Java. The Real-Time for JavaTM Expert Group.
9) Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying gc without stopping the world. *Joint ACM Java Grande - ISCOPE 2001 Conference*, 2001.