リアルタイム環境に適したガベージコレクション API の実現について

権藤 克彦*，八十島 利典*，喜多山卓郎†

平成 12 年 3 月 12 日

概 要

本論文では Java のガベージコレクション API(GC API) を提案する．この GC API を用いると，プログラマはリアルタイム制約を満たすためにアプリケーション毎に GC の動作を変更することが可能となる．RT-Mach 上で動作する Java VM にこの GC API の一部を実現した．

キーワード：ガベージコレクション，リアルタイム，Java，API，Kaffe OpenVM

1 はじめに

プログラミング言語 Java[5] は高い移植性と再利用性などの良い性質を持つため，組み込みシステムやリアルタイムシステムにまで応用範囲が広がりつつある．このため，Java のリアルタイム拡張仕様 [6][7] の作成が進められているが，ソフトウェアの保守性を保ちつつ，これらのシステムが要求する性能 (例えばリアルタイム性) を幅広く満たすことは難しく，まだ実現には至っていない．

Java のリアルタイム拡張において解決すべき問題の 1 つに「リアルタイム処理に適したガベージコレクタ (以下 GC) の構築」がある．GC を持つシステムでは，最大応答時間の保証，GC の処理時間の予測，ヒープ残量の保証とメモリ割り当てに必要な時間の予測などが一般に難しいため，リアルタイム制約を満たしにくいからである．

本論文では，リアルタイム処理[1]に適した GC の機能として，Java のガベージコレクション API(GC API) を提案する．この GC API を用いると，プログラマはリアルタイム制約を満たすためにアプリケーション毎に GC の動作を変更することが可能となる．本来，GC はアプリケーションプログラマに対して透明，つまりメモリ管理を意識させないことが重要であるが，我々は次の理由から GC API が有用であると主張する．

- リアルタイム用の GC アルゴリズムの改良や GC のチューニングだけではリアルタイム制約の充足は難しい．GC の性能はアプリケーション，動作環境，GC のパラメータによっても大きく変化するからである．
- アプリケーションや動作環境ごとに自動的に GC の動作を調整できれば理想的だが，現在の技術では難しい．現在提案されている自動スケジューリングは，処理時間とデッドラインが既知といった強い仮定が必要である [13]．
- 多くの場合，アプリケーション開発者はどのように最適化すれば良いかを知っている．また，最適化が必要な部分はコードの一部にすぎないことも多い．それゆえ，GC API があれば，保守性を保ったまま，そのアプリケーションに特化した GC の動作をチューニングしやすい．

また，細部までは決まっていないものの Java のリアルタイム拡張 [6] に GC の動作変更や状態取得のための API が提案されていること，リアルタイム用ではないがカスタマイズ可能な GC フレームワーク [8] が提案されていること，も我々の主張を裏付けている．

我々が提案する GC API は，次の 3 種類である．

*北陸先端科学技術大学院大学 情報科学研究科
†ソニー株式会社インフォメーション＆ネットワーク研究所コンピュータシステムラボラトリ

[1] 本論文ではソフトリアルタイム処理 (リアルタイム制約の充足を完全には保証しない) を仮定する．

- GC の実行間隔や優先順位の指定
- GC の起動抑制・許可
- メモリ予約

もちろん，これ以外にも多くの有用な APIがあるが，我々は実現と実験を優先すべきと考えて，あえてシンプルな API としている．

我々はこの GC API の一部を実現した．GC APIの実現に用いた環境は，Kaffee OpenVM(kaffe-1.0.b1)[10] を RT-Mach マイクロカーネル [11] 上に移植した kaffe-1.0.b1-rt[9] である．kaffe-1.0.b1-rt は RT-Mach のリアルタイムスレッドをネイティブスレッドとしており，デッドラインミスが起こった時に動作するデッドラインハンドラを記述できることが特徴である．

GC APIを実現する上で，kaffe-1.0.b1-rt の GC に次の修正を施した．

- 湯浅のスナップショットアルゴリズムを用いて並行 GC とした．
- write-barrier はメモリ保護機構と RT-Mach の例外処理機構を用いて簡易に実現した．
- ヒープ残量だけでなく，一定時間ごとに GC を動作させるためにタイマスレッドを追加した．

本論文の以降の構成を示す．2 章では我々が提案する GC API を説明し，3 章では kaffe-1.0.b1-rt 上への実現方法を述べる．4 章では実現に関する議論を行い，5 章では関連研究，6 章では結論と将来の展望を述べる．

2 GC APIの設計

本節では，本研究で提案する GC API とその方針とを説明する．

2.1 方針

我々の考える GC API の位置づけを図 1 に示す．

- 現在の Java の範囲内でできるメモリ管理 (例えば，オブジェクトプール，`static` 変数の多用) よりもきめ細かく GC の挙動を制御したい．例えば，オブジェクト以外の，JVM で使用されるメモリ (例えば JIT に使われるメモリ) も制御したい．
- ハード リアルタイム制約を満たすには，おそらくハードウェア支援 GCが必要である．GC API はそこまではカバーしない．
- API なしの GC とはカバーできる範囲が異なるはず．実際にどのくらい異なるかを知りたい．

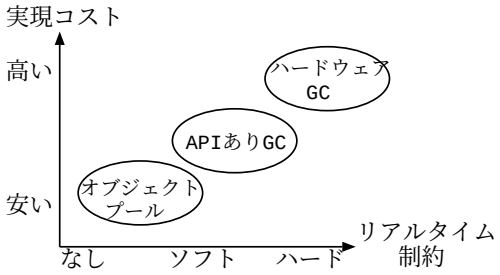

図 1: GC の実現コストとリアルタイム制約の関係

ここで，我々は次の方針でリアルタイム環境に適した GC API を模索したいと考えている．

> **方針:** 素朴で小さな API のセットで良いから，まず実現して，動作実験により有用性を確認する．

この理由を次に挙げる．

- 単純な GC API(パラメタ調整，GC の起動抑制，メモリ予約) でもかなり有効と予想している．
- Java 言語と処理系全体へのリアルタイム拡張仕様は重要だが，規模が大きいため，GC API の有用性をなかなか確認できない．
- GC API の抽象化 (Java 言語への拡張を含む)，異なる GC API の記述力の比較も重要だが，それよりも先に GC API というアイデア自体の有用性を確認したい．

2.2 提案する GC APIの概要

表 1 に GC API の一覧を示す．主に次の 3 つの機能がある．

- GC の実行間隔や優先順位の指定

  この API は，各アプリケーションに適した実行間隔を時間単位で指定したり，優先順位を指定する．Timed-GC[1] のアイディアを採用している．

- GC の起動や preemption の抑制・許可

  本論文では，(mark-sweep 方式を仮定して)「GC の初期化，マーク，スイープ」全体を GC の起動と呼ぶ．

  GC に preempt されたくない部分では，GC の preemption を抑制する．また，write barrier のオーバーヘッドを無視できないほどリアルタイム制約が非常に厳しい部分には，新たな GC の起動自体を抑制して，デッドラインミスを回避する．

- メモリ予約

  GC を持つ実行環境では，メモリ割り当て時に，GC を少し実行させたり，GC に必要な処理 (割り当てたメモリのマーク領域の確保と初期化など) を行うため，一般にオーバーヘッドがある．また，ヒープ残量不足で，GC の終了待ちが必要なこともある．つまり，ヒープ残量の保証やメモリ割り当てに必要な時間の予測が難しい．

  このため GC を経由せずにメモリを予め割り当てておきたいことがある．この機能をメモリ予約と呼ぶ．メモリ予約の API は，GC を経由せずに明示的に動的メモリ管理をさせることで，メモリ残量を保証し，メモリ割り当てに必要な時間をより小さく，より予測可能にする．

以上の GC API では，全てのユーザスレッドに対して，GC スレッドおよび GC が管理するヒープが 1 つしかないと仮定している．このため，あるスレッドが GC API を利用すると，その影響は他のスレッドにも及ぶ．つまり，GC API はどのスレッドからでも使えるが，その使用が衝突しないことは，プログラマが保証しなくてはいけない．

以下ではこの GC API の簡単な使用例と説明を示す．

2.3 GC の実行間隔と優先順位の指定

```
// 100ms 毎に GC を 10ms 実行すると設定
GC.setInterval(100000, 10000);

// 優先度を最大に設定
GC.setPriority(1);
```

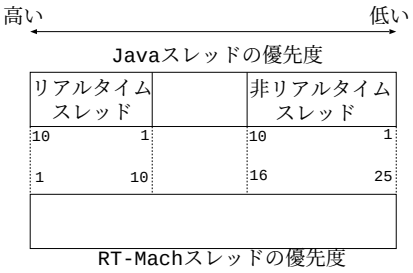

図 2: スレッドの優先度

我々が実装に用いた JVM である kaffe-1.0.b1-rt[9] では，Java のネイティブスレッドとして，RT-Mach スレッドを用いている．ここで，`GC.setPriority(1)` がセットする優先度は，実装に依存して，RT-Mach スレッドの優先度であることに注意して欲しい (図 2)．GC API の実装依存性については 4 節でふれる．

2.4 GC の起動抑制・許可

```
// GC が起動中でなければ，アトミックに
// GC の起動を禁止して
if (GC.testAndSetInvokable(false)) {

    // ここにリアルタイム制約の
    // 厳しい処理を書く

    // GC の起動禁止を解除
    GC.setInvokable(true);
}
```

2.5 メモリ予約

```
// 50KB のメモリを予約
int mem_id = GC.ReserveMem(50);

// 予約したメモリからの割り当て開始
```

実行間隔の指定	start マイクロ秒毎に period マイクロ秒だけ GC を実行する． public void GC.setInterval (int start, int period) public int[] GC.getInterval ()
優先順位の指定	GC の優先順位を pri とする．範囲は 1 〜 25 で，1 が最も優先順位が高い． public void GC.setPriority (int pri) public int GC.getPriority ()
GC 起動の抑制・許可	b が false ならば，新たな GC の起動 (GC の初期化からの実行) を抑制する．すでに起動中の GC を終了させる効果は無い． public void GC.setInvokable (boolean b) public boolean GC.testAndSetInvokable (boolean b) public boolean GC.isInvokable () public boolean GC.isInvoked ()
GC の preempt の抑制・許可	b が false ならば，GC スレッドに preempt しない．すでに実行中の GC を終了させる効果は無い． public void GC.setPreemptable (boolean b) public boolean GC.isPreemptable ()
メモリ予約	GC の処理対象とならない size KB のメモリを割り当てて，予約番号を返す． public int GC.ReserveMem (int size)
予約の行使	予約番号 id のメモリを使用するか否かを b で指定する． public int GC.ProvideMem (int id, boolean b)
予約の開放	GC.ReserveMem() で予約した予約番号 id のメモリを開放する． public void GC.FreeMem (int id)

表 1: ガベージコレクション API 一覧

```
GC.ProvideMem(mem_id, true);

// 予約したメモリ上にインスタンスを生成
Foo foo = new Foo();

// 予約したメモリからの割り当て終了
GC.ProvideMem(mem_id, false);

// 予約したメモリの開放
GC.FreeMem(mem_id)
```

予約メモリからの割り当て中は，JVMで必要とされる全ての動的なメモリ (例えばJITやクラスロードに必要なメモリ) が予約メモリから高速に割り当てられる．

その反面，メモリ予約には以下の制約があり，プログラマがこれを保証しなくてはいけない．

- 予約したメモリ量が不足しないこと．
- foo が参照するオブジェクトをGC.FreeMem(mem_id) 以降で参照しないこと．GC API のセマンティクスとして，foo が参照するオブジェクトの生存期間はGC.FreeMem(mem_id) までであり，dangling pointer の原因となるから．
- 予約領域のオブジェクトが，GC 管理下のオブジェクトを参照しないこと．

メモリ予約のセマンティクスとして，以下の考慮すべき点がある．

- 「GC.FreeMem(mem_id) 以後はGCの対象となり，参照は可」というセマンティクスにすることもできるが，kaffe-1.0.b1 の場合はconservative GCなので，これは難しい．
- 上記の制約を排し，メモリ予約を安全にすることも原理的には可能だが，安全性と効率はトレードオフであり，現在は効率を重視している．この点は研究課題として議論の余地がある．
- 予約メモリ上に割り当てられた，オブジェクト以外のメモリがGC.FreeMem(mem_id) 以降で参照されないことをプログラマは保証できないので，GC.FreeMem(mem_id) 時にこれらは解放されないとしている．

3 GC APIの実装

3.1 kaffe-1.0.b1{-rt} のGCの概要

まずGC APIの実装のベースとして用いているkaffe-1.0.b1 と kaffe-1.0.b1-rt について説明する．

kaffe-1.0.b1 はC言語で実現されており，仮想マシン部分が約20,000行，GC部分が約2,000行という規模である．kaffe-1.0.b1 から kaffe-1.0.b1-rt を実現する際に，GCには本質的に変更は加えられなかった．kaffe-1.0.b1{-rt} のGCの特徴は以下の通り．

- mark and sweep — 無条件で必要とされるオブジェクトの集合をルートという．mark and sweep は，ルートから到達可能なオブジェクトにマークをつけて，マークされなかったオブジェクトを回収するGCの手法である．
- conservative — ポインタとデータの区別ができない場合があり，かつ，その場合にポインタとして扱うGCをconservative GCという．ルートの一部であるネイティブメソッドのスタックの扱いを簡単化するため，kaffe-1.0.b1 のGCはconservativeとなっている．メモリリークが起きること，オブジェクトを別アドレスに移動できないため，copying や compaction が出来ない，という欠点がある．
- 2-level allocation — conservative GCの欠点であるメモリのフラグメンテーションを軽減するために，動的なメモリ割り当てを2階層にする手法を2-level allocationという．下位レベルではシステムからまとまった大きさ (例えば1MB) で動的にメモリを確保し，上位レベルではそれをブロック毎に固定サイズ (例えば16B, 32B, 64B,...) を決めて，GCにメモリを供給する．

3.2 GC APIの実現の概要

表1のGC APIを実現するために，kaffe-1.0.b1{-rt} のGCに以下の修正を行った．

- 湯淺のスナップショットアルゴリズム [2][12] を用いて並行GCとした．
- write-barrier はメモリ保護機構と RT-Mach の

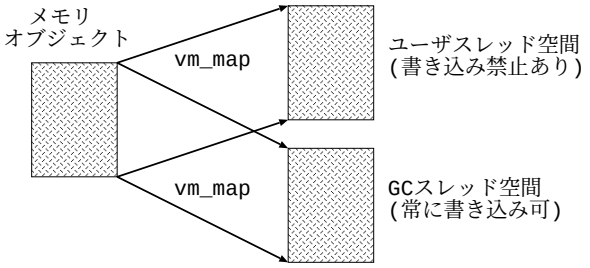

図 3: vm_map によるメモリ空間の分割

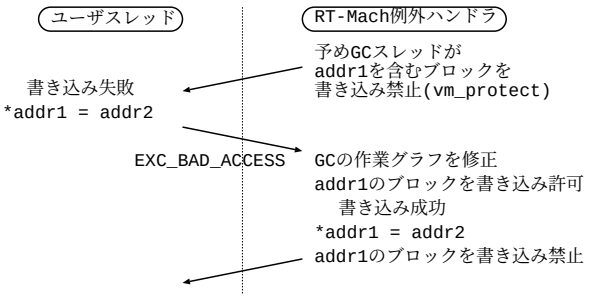

図 4: write-barrier の概要[2]

例外処理機構を用いた OS 支援方式で簡易に実現した.

- ヒープ残量だけでなく，一定時間ごとに GC を動作させるためにタイマスレッドを追加した.

湯淺のスナップショットアルゴリズムでは，GC がマーキング中でも，ユーザプログラムが動作する．この際に，ユーザプログラム中でポインタの書き込みを行うと，GC の作業グラフとの一貫性が崩れることがある．そこで，GC のマーキング中は，ポインタの書き込み時に，GC の作業グラフも変更して一貫性を保つ必要がある．このポインタ書き込み時に必要な一貫性を保つ処理の仕組みを write-barrier という．以下では，我々の write-barrier の実現方法について説明する.

まず，vm_map を用いて，同じメモリオブジェクトを 2 つのメモリ空間としてマップする．1 つはユーザスレッド用であり，GC のマーキング中は書き込み禁止にされる．もう 1 つは GC スレッド用であり，常に書き込み可能である (図 3)．これにより，マーキング中は GC 以外には書き込み禁止となる.

次に，GC のマーキング中にユーザスレッドがメモリへの書き込みを行うと，例外 EXC_BAD_ACCESS が発生し，例外ハンドラが起動される．例外ハンドラ中で，GC の作業グラフの修正を行う．つまり，この例外ハンドラが write-barrier として働く (図 4).

3.3 プログラム例

残念ながら，デバッグやチューニングが不十分なため，GC API を用いて有意な差が出るプログラムをまだ動作できていない．ここでは，GC API を用いていないが，(並行 GC なので) デッドラインミスが少なくなるプログラム例を示す.

```
import kaffe.rt.*;
class myProgram {
    public static void main(String args[]) {
        // デッドラインハンドラ (スレッドの一種)
        // を作成して起動
        MyHandler handler = new MyHandler();
        handler.setPriority(10);
        handler.start();

        // タイミング属性の設定
        Time period   = new Time(1, 0); // 1 秒
        Time deadline = new Time(0, 35000000);
                                    // 35 ミリ秒
        Time start    = new Time(0, 0); // 0 秒

        MyRtThread rtThread = new MyRtThread();

        // スレッドの属性を設定
        rtThread.setAttr(1,          // 優先度
                         start,     // 開始時間
                         period,    // 周期時間
                         deadline); // デッドライン
        // ハンドラオブジェクトとメソッド名の指定
        rtThread.setHandler(handler, "Handler");
        rtThread.start();
    }
}

class MyHandler extends RtHandler {
    public void     Handler(RtThread Th) {
        // デッドラインミスしたスレッドを再開
        Th.resume();
    }
}

class MyRtThread extends RtThread {
    public void run() {
        System.gc(); // 周期的に GC を起動
    }
}
```

[2] addr1 自体を再び書き込み禁止にする必要は無い．しかし書き込み禁止の単位がブロックであり，addr1 の周辺は書き込み禁止にする必要があるため，addr1 のブロックを書き込み禁止している.

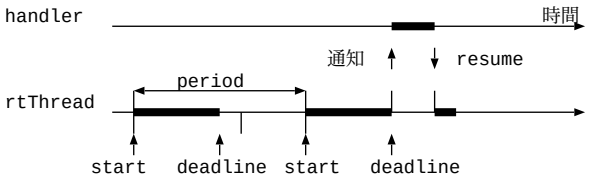

図 5: リアルタイムスレッドとデッドラインハンドラ

このプログラム中のリアルタイムスレッド (rtThread) は，周期的に動作し，指定したデッドライン (35ms) を満たせないと，デッドラインハンドラのメソッド Handler を起動する (図 5)．このプログラムを kaffe-1.0.b1-rt で動作させると，35ms では GC が終了しないため，10 回の繰り返しで 10 回のデッドラインミスが起きた．我々が実現した並行 GC では，System.gc() はブロックしないので，デッドラインミスは 1 回のみであった．この 1 回はクラス System のロードが原因である．

4 議論

4.1 RT-Mach 例外ハンドラによる write-barrier の実現

実現は容易である

UNIX のシグナルハンドラとは異なり，RT-Mach 例外ハンドラには引数として，例えば，図 4 の addr1 が渡される．また，ソフトウェア方式 (以下参照) で必要となる，VM 中への write-barrier 埋め込むが不要のため，3.2 節の方式による write-barrier の実現は比較的容易であった．

オーバーヘッドが大きい

3.2 節の write-barrier は実現が容易だが，残念ながら，この write-barrier のオーバーヘッドは大きすぎて (約 130μs [3])，あまり実用的ではない．オーバーヘッドが大きい原因は主に以下である．

- 例外ハンドラを起動するオーバーヘッド
- 図 4 でのアドレス addr1 のチェック (conservative GC なので必要)
- addr1 の書き込み許可/禁止

今回の実装では実現の容易さを重視したため，この実現法を採用したが，オーバーヘッドが小さくなる実現法を採用すべきである．しかし，最終的にはソフトウェア方式 (以下参照) を採用すべきだろう．

ソフトウェア方式の write-barrier の実現に役立つ

ソフトウェア方式は，ポインタの書き込み (あるいは読み込み) のコードをコンパイラが生成する時に write-barrier 処理 (あるいは read-barrier 処理) のコードを付け加える方式である．Java の場合は，JIT コンパイラだけではなく，インタプリタやネイティブメソッド中の全てのポインタ書き込みに対しても，write-barrier 処理のコードを埋め込む必要があるので，実現は煩雑となる．我々が今回実現した OS 支援方式は，ソフトウェア方式を実現した時に write-barrier 処理のコード埋め込み忘れを実行時に検出できるので，ソフトウェア方式の write-barrier 実現を容易にするというメリットがある．

4.2 GC のコードは保守性が低い

今回の我々の GC API の実現の作業では，予想以上にデバッグ作業が困難であり，多くの時間を占めている．このため我々は「GC のコードの保守性は低い」と強く感じている．この原因は (我々のスキルの低さもあるが) 以下にあると分析した．

- アルゴリズムの内容に比べて，コード量が大きい．つまり，非機能的要件を実現するコード部分が多い．言い換えると GC はチューニングの塊である．
- コード部分同士の結合度が大きく，コード変更の影響が広範囲に渡りやすい．
- GC の性質上，バグの原因を特定しにくい．GC が誤って使用中のオブジェクトを回収すると dangling pointer の原因となる．これは GC のバグが原因だが，ユーザスレッド中でのメモリ不正アクセスとして現れる．また，メモリリークは発見そのものが難しい．

[3] Pentium Pro, 200MHz, 64MB メモリ, RT-Mach Lites MKNG008 での測定.

- 並行GCなので，並行プログラムとしてのデバッグの難しさもある．

このため，GCの効率を犠牲にせずに，保守性を高く保つ技術，例えばGC用のアスペクト指向言語，が必要だと考えている．

4.3 GC APIは実装に強く依存する

例えば，`GC.setPriority(prio)`で指定する優先度は，RT-Machの優先度としており，実装プラットフォームに依存している．また，mark-sweep方式で問題となる一括ルート挿入のオーバーヘッド見積りを返すGC APIを用意した場合 (今回は与えていないが)，このAPIはGCのアルゴリズムや実装に依存している．

それゆえ，一般的にAPIは実現に対して独立であるべきだが，GC APIはGCの実装に強く依存するし，依存すべきであると我々は考えている．GCの実装ごとに，調整できるパラメタや，GCの動作を変更する操作の集合が異なるので，無理なAPIの統一は有効なチューニングを妨げると考えるからである．GC APIをうまく依存/非依存で分離して，見通しは良くする必要があるが，本論文ではこの問題を無視している．

5 関連研究

5.1 RTJEGのリアルタイム仕様拡張

RTJEG(Real-Time for Java Experts Group)[6]のリアルタイム仕様拡張は，54ページの仕様書であり，本研究で提案したGC APIと類似のAPIを持つ．

- クラス`GarbageCollector`は，

  ```
  void setPreemptionThreshold (int t)
  void setReclamationRatio (int ratio)
  ```

  というGCのパラメタを指定するAPIを持つ．また，

  ```
  int getBarrierOverhead ()
  int getPreemptionLatency ()
  ```

  など，GCのオーバーヘッドの量を取得するAPIも用意されている．

- GCの起動の抑制に相当するAPIはない．
- メモリ予約に類似した機能として，`ImmortalMemory`，`ScopedMemory`，`VTMemory`というクラスが定義されている．

5.2 J consortiumのリアルタイム拡張仕様

J consortiumのリアルタイム拡張仕様[7]は，128ページの仕様書であり，本研究で提案したGC APIと類似のAPIを持たない．

この仕様書では，従来のJavaをBaseline Javaと呼び，この仕様書で定義するJavaをCore Javaと呼んで区別している．両者は異なるクラスツリーを構成する．また，実行環境においても両者は異なるヒープを用いると規定されており，Core Javaのオブジェクトが使用するヒープはプログラマによる明示的な制御によって管理される仕様となっている．これゆえ，Baseline JavaからCore Javaへのメソッド呼出し，および逆のメソッド呼出しは強い制限を課せられている．この点で，RTJEGとはかなり異なる性質の仕様となっている．

6 おわりに

本論文では，リアルタイム処理に適したGCの機能として，Javaのガベージコレクション API(GC API)が有用であると主張した．また，kaffe-1.0.b1-rtに基づく実装方法を示し，基本的な実現可能性を確認した．

本研究はまだまだ中途段階にあり，本研究の有用性は確認できていない．GC APIの実装を完成させた上で，GC APIによるリアルタイム性の改善の実例を示す必要がある．

謝辞

情報処理振興事業協会 (IPA) 独創的先進的情報技術に係わる研究開発「組み込みシステム用基盤ソフトウェアSMAFの研究開発」の1テーマとして，本研

究を行っています．本研究を進めるにあたり，SMAF プロジェクトの皆様から多大な助言を頂きました．また，査読者の皆様には誤りの指摘や有用な助言を頂きました．ここに感謝の意を表します．

参考文献

[1] Ito, T. and Sasaki, H., Timed-GC for a real-time Lisp system, Proc. ACM SIGPLAN Workshop on Langauges, Compilers and Tools in Real-Time Systems, 1997

[2] Richard Jones and Rafael Lins, Garbage Collection, John Wiley & Sons, ISBN 0-471-94148-4, 1996

[3] 岩井 輝男，中西 正和，Snapshot 型並列 GC におけるルート挿入時間の削減，情報処理学会論文誌プログラミング,Vol40,SIG4(PRO3)

[4] 組み込みシステム用基盤ソフトウェア SMAF の研究開発ホームページ http://www.mkg.sfc.keio.ac.jp/smaf/

[5] James Gosling, Bill Joy, and Guy Steele.，村上 雅章 訳，The Java 言語仕様. アジソン・ウェスレイ・パブリッシャーズ・ジャパン, 1997.

[6] The Real-Time for Java Experts Group (RTJEG), Real Time Specification for Java ver 0.8.1, 1999, http://www.rtj.org/rtj.pdf

[7] J Consortium, Real-Time Core Extension for the Java Platform (Draft 1.0.2), http://www.j-consortium.org/spec.PDF

[8] Giuseppe Attardi, Tito Flagella and Pietro Iglio, A Customisable Memory Management Framework for C++, Software Practice and Experience, vol. 28(11), pp. 1143–1183, 1998.

[9] A. Miyoshi, T. Kitayama, H. Tokuda: Implementation and Evaluation of Real-Time Java Threads, 18th Real-Time Systems Symposium,pp. 166–175, 1997.

[10] Transvirtual Technologies, Inc., Kaffe OpenVM, http://www.transvirtual.com/

[11] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. Proc. USENIX 1st Mach Workshop, October 1990.

[12] Taichi Yuasa, Real-time garbage collection on genral-purpose machines, Journal of Software and Systems, 11(3), pp.181-198, 1990.

[13] Roger Henriksson, Predictable Automatic Memory Management for Embedded Systems, OOPSLA'97 workshop on Garbage Collection and Memory Management, 1997.

[14] 風間 一洋，Java の最新技術動向，Unix Magazine 1 月号，pp.36–49, 2000.