Vol.2012-SE-176 No.2
Vol.2012-EMB-25 No.2
2012/5/21

移動のないIncremental GCにおけるリアルタイム性評価

養安 元気[1,a)]　菅谷 みどり[†1]　井出 真広[1]　倉光 君郎[1]

概要：現在, スクリプト言語処理系に Garbage Collection(以降 GC) が組み込まれていることが一般的である. GC による自動メモリ管理は開発を容易にすることから，最近では組込みシステムへも適用領域が拡大している. しかし, GC はプログラム実行中にメモリ管理を行うため, リアルタイムシステムにおいては, その際のオーバーヘッドと, mutator の最大停止時間が問題となる. 本研究では、mutator の最大停止時間を短縮する目的で Incremental GC を, またオーバーヘッドの逓減のために複数 bitmap を用いた Non-moving 世代別 GC の両者をスクリプト言語 konoha に組み込む. Non-moving 世代別 GC はオブジェクトの移動を減らし, GC のオーバーヘッドを逓減させる効果がある. 評価実験では, ハードリアルタイムを提供する ART- Linux の API に konoha 言語の API をマップし, アプリケーションの実行を通じた GC の評価を行った. その結果, mutator の停止時間が従来の konoha 言語の Mark and Sweep GC と比較して，2.66%, 停止時間のジッタを示す標準偏差が 1.44 %まで減少し，実用的なリアルタイムシステムへの適用可能である事を示した.

キーワード：ガーベージコレクション，リアルタイムシステム，RT-Linux

Non-Moving Incremental Garbage Collections in Realtime System.

Motoki Yoan[1,a)]　Midori Sugaya[†1]　Masahiro Ide[1]　Kimio Kuramitsu[1]

Abstract:
Recently, scripting language processors include Garbage Collection(GC). Automated memory management make the development efficent, so that even if the embedded systemm apply for it. However, a propblem arouse in real-time application that suffers their overhead. Moreover GC manages chanks of memory during a program execution, it causes the maximum stop time. It will not suitable for real-time applications. To reduce the stop time and overhead, we propose Incremental GC and its extended generational GC that reduces also the search time for object and overhead. We implemented these GCs on static typing konohaing language, and evaluate on ART-Linux that provides hard real-time system. We bind the API libraies and develop applications. The evaulation shows the effectiveness of our proposal.

Keywords: Garbage Collection, real-time system, ART-Linux

1. はじめに

現在スクリプト言語処理系に, Garbage Collection (以降 GC) が組み込まれていることが一般的である. GC が提供する自動メモリ管理によって, スクリプト言語の開発効率が高まることから, 組込みシステムなどを含め, その適用領域は拡大している.

一方 GC は自動的にメモリ管理を行う反面, GC が起動して動作するためのオーバーヘッドや, それによるプログラムの実行停止時間が問題となる. 特に, リアルタイムシステムにおいては, プログラムの実行を阻害しないよう, GC による実行停止時間の最悪値 (最大停止時間) をできるだけ小さくする事が重要な課題となる.

一般的に用いられる Mark and Sweep GC の場合, マークフェーズの処理時間は, ライブオブジェクトの数に依存し, スィープフェーズはヒープサイズに依存する. 特にライブオブジェクトの数は, プログラムにより異なるため, GC 動

[1] 横浜国立大学大学院 工学府物理情報工学専攻
Yokohama National University, Kanagara, 240–8501, Japan
[†1] 現在，横浜国立大学
Presently with Yokohama National University, MICT, Kanagara, 240–, Japan
[a)] myoan@konohascript.org

Vol.2012-SE-176 No.2
Vol.2012-EMB-25 No.2
2012/5/21

作によるプログラムの停止時間は不確定となる.

これに対して, Incremental GC によりリアルタイム性能を向上させる手法が提案されている [1][5]. インクリメンタル GC は, ユーザプログラムを実行するスレッド (mutator) の実行中に, GC を行うスレッド (collector) 処理を時間的に分散させる事で, 毎回の GC による mutator の停止時間を短縮する.

また，別のリアルタイム性の阻害要素として, スクリプト言語処理系によっては,GC 時のポインタ参照コストの問題がある. 単純な Mark and Sweep GC の場合，Mark 処理は生きているオブジェクトの数に比例して，Sweep 処理は言語処理系が確保しているメモリ領域の大きさに比例して，処理にかかる時間が増加する．Mark 処理のトレース時間の短縮には, 様々な手法が提案されているが, 中でも世代別 GC は, 世代別を用いていない GC に比べ, minor GC に要する時間が非常に小さい特徴がある.

本研究では, 実用的なリアルタイムシステムへスクリプト言語を適用することを目的とし，それにあわせた GC の開発を行う．具体的には，Mark 処理のトレース時間の短縮，Sweep 処理の操作時間の短縮を目的とし，静的型付けを持つ konoha [2][3] 言語処理系に bitmap を利用した Mark and Sweep GC(Non-moving bitmap GC) を実装した．さらに，Non-moving bitmap GC を世代別 GC と組み合わせる事により, トレース対象の削減を行った. また, mutator の最大停止時間を短縮する目的でインクリメンタル GC の実装の一つである snapshotGC を実装した. この際に, bitmap をマークとアロケート用に別々に準備する事で, マークを中断してアロケーションが発生しても, bitmap の状態の保持を可能とした.

本論文では, 実際にロボットなどで広く利用されているハードリアルタイム上でを提供する ART-Linux 上でアプリケーションを開発し, 評価を行った. konoha には, ART-Linux が提供するリアルタイムアプリケーション向けの API のためのライブラリを実装し， ライブラリを利用した実際のアプリケーションを開発した.

評価では, mutator の最大停止時間を Mark and Sweep GC と比較して 2.66%, 停止時間のジッタの標準偏差を 1.44%に抑え，リアルタイムアプリケーションでの有用性を確認した.

本論文では，まず，第 2 節で konoha 言語と課題について述べる. 第 3 節で Non-moving 世代別 Incremental GC の設計を述べた後，第 4 節にてその実装について述べる．第 5 節で実装した処理系の性能測定とその評価を行う．第 6 節で関連研究を述べ，第 7 節にてまとめる.

2. スクリプト言語への Non-moving bitmap GC の適用

2.1 特徴

現在，スクリプト言語処理系は,GC の機能を備えている. 言語の設計方針により優先するべき性能が異なるが, 主にスループット, レスポンス性能, メモリ使用効率などの達成を目的とした GC 手法が提案されている. 我々が開発する konoha 言語では, 主にスループットの向上を目的とした Non-moving 世代別 GC[4] を実装している.

Non-moving bitmap GC は copying GC と比較して, copy のコストが少なく, フラグメンテーションを発生しないメリットがある. また, 世代別 GC は, それを用いていない GC に比べ, minor GC に要する時間が非常に小さい特徴がある.

特にオブジェクト指向型言語の場合，スループットの向上には世代別 GC が有効であるとされる．これは，オブジェクト指向型言語の，生成されたばかりのオブジェクトはすぐに死亡する (解放される) という特性から，長く生きる (tenure) オブジェクトよりも若い (young) オブジェクトを優先的に GC することで，トレース時間を減らせるためである．young オブジェクトはある程度の GC を生き延びると，tenure オブジェクトに昇格する．世代別 GC の多くはオブジェクトの情報を別領域に保存し，以前のオブジェクトを解放することで昇格を実装している.

世代別 GC の実装の多くは，young オブジェクトと tenure オブジェクトを別の領域に分けて世代を区別しており, 昇格 (young オブジェクトが tenure オブジェクトとなる) の際に，オブジェクトの移動を行うものとなっている. この際,konoha 言語では, 無駄にオブジェクトを移動させない (Non-moving) の方針を取ることで, スループットをさらに向上させる実装を行っている.

本節では, まず始めに, Non-moving bitmap GC と, 世代別 GC を組み合わせた設計の内容について述べる.

2.2 Non-moving bitmap GC

Non-moving bitmap GC [4] は Mark-and-Sweep GC であり，(1) 効率的なアロケーション, (2) フラグメントの抑制, (3) 遅延スイープによる GC にかかるコストの削減 (4) オブジェクトの移動のない世代別 GC の実装の特徴を持つ. 以下に，アロケーションアルゴリズムを中心とした設計を述べる.

2.2.1 メモリ構造

Non-moving bitmap GC のメモリ構造を図 1 に示す. Non-moving bitmap GC では, ヒープ領域の管理のために， ヒープマネージャとなる親を頂点として, 2 のべき乗毎のサイズに分けたサブヒープを用意する. 一般的に, サブヒープは 8bytes から 4Kbytes までを利用するが, konoha 言語では, オブジェクトの最小サイズを 64bytes としているため, 64bytes

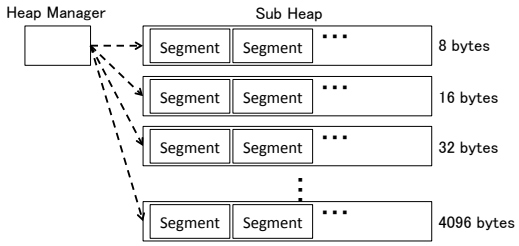

図 1 メモリ構造

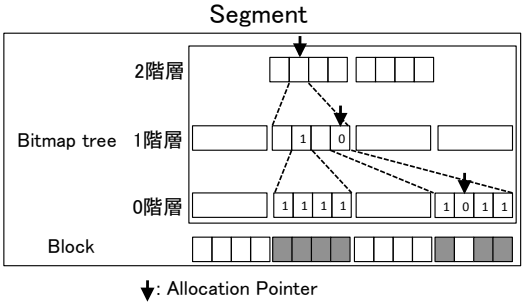

図 2 Segment 構造体

から 4Kbytes までのオブジェクトを格納するサブヒープを用意した.

なお，4Kbytes を超えるオブジェクトを生成する場合は，個別に malloc() によりメモリ操作を行い，konoha で新しく作成されたオブジェクトのアロケートを行う．サブヒープはそれぞれ，セグメントと 2.3 で後述するアロケーションポインタを持ち，このセグメントが実際にオブジェクトを格納する領域を保持している．セグメント構造体に含まれる要素を以下 (1) に示した. ここでは, 128Kbytes を 1 セグメントとしている.

$$Segment = (livecount, blocks, bitmaptree, RememberedSet) \quad (1)$$

図 1 に構造を示した. 図に示したように, Segment 構造体は, bitmap tree とオブジェクトの格納領域 (blocks) を保持し，bitmap の bit と各 block は 1 対 1 対応する. livecount とは，現在そのセグメントで使用している block の数とであり，Remembered Set は後に実装する, 世代別 GC の時に利用する.

セグメントは固定長であり，サブヒープは複数のセグメントをリスト構造で連結して保持している．また，block の先頭には, オブジェクトを格納する代わりに，blocks を保持するセグメントのポインタを格納する. これにより, オブジェクトからセグメントへの参照と, オブジェクトからそれに対応する bitmap への参照が可能となる.

2.3 アロケーション アルゴリズム

Non-moving bitmap GC では GC 時のマークに用いる bitmap をアロケーションにも利用する．アロケーションで利用する際に，常に，bitmap の先頭から空いている bit を操作して探すのはコストが高いため，図 2 のように，bitmap を木構造として構成した (以降，bitmap tree と呼ぶ), 第 0 階層の bit はオブジェクトを格納する block と 1 対 1 対応する．同様に第 1，第 2 階層の bit は一階層下の bitmap と 1 体 1 対応する．対応する下の階層の bit が全て 1 である場合にのみ，その bit を 1 とする．各階層の bit は 64bit OS の場合，64 個の (32bit OS の場合，32 個の)1 階層下の bit と対応する．マークの場合，bitmap tree の第 0 階層のみを用いて block に格納されたオブジェクトの生死判定を bit(真偽値) で判定する．また，アロケーションで利用する場合，図 2 での bitmap tree の第 0 階層の左端から右へ順にアロケートし，満杯になったら一階層上の bitmap を右に移し，対応する第 0 階層へアロケートを行う.

konoha ではアロケートの効率化のため，サブヒープは，次にアロケートされる各階層の bit の位置を記憶した Allocation Pointer を持つものとした. Allocation Pointer 構造体に含まれる要素を以下 (2) に示す.

$$AllocationPointer = (segment, bitmappointer, blockpointer) \quad (2)$$

サブヒープは 3 のような構造になっており，それぞれの要素は，Allocation Pointer の他に，そのサブヒープが格納するオブジェクトの大きさを表す class_size, サブヒープに空きがあるかどうかを判定する isFull, 使用しているセグメントのリスト segList, 空いているセグメントのリスト freeSegList, segList の現在の数と，最大の数を示す segList_size, segList_max となっている.

また，オブジェクトはアロケートされる前に，前回そのオブジェクトを解放する．このように deferred sweep にすることで，Sweep 時にメモリ全体を走査するコストを削減する.

$$SubHeap = (Allocationpointer, class_size, isFull, segList, freeSegList, segList_size, segList_max) \quad (3)$$

segment は次に空いている block がある segment を指しており，bitmap poiner, block pointer はそれぞれ，そのセグメントにおける，次に空いている場所を指している．この Allocation Pointer が指しているブロックが空いていなければ，一階層上の bitmap を見て，空いている bit があれば，その bit と対応する一階層下の bitmap を走査する．空いているオブジェクトが見つかると，そのオブジェクトを解放し，初期化した後，mutator に渡す.

Vol.2012-SE-176 No.2
Vol.2012-EMB-25 No.2
2012/5/21

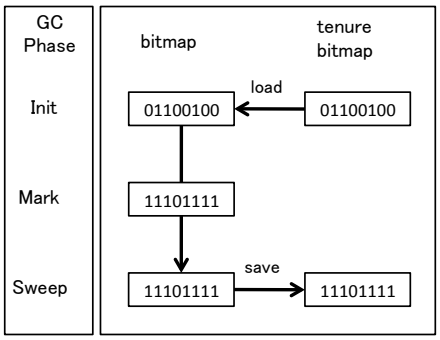

図 3 世代別 GC の bitmap tree の遷移図

2.4 Non-moving bitmap GC の問題点

Non-moving bitmap GC は, Mark and Sweep GC であるが, Sweep はツリーをたどる遅延 Sweep である. Sweep フェーズでは，block の空いているセグメントをリストで連結し，サブヒープに freeSeglist として保持させる．次回のアロケーションの時，セグメントがオブジェクトで満たされた時，freeSegList からセグメントを取り出すようになっている．そのため, アロケーションのたび freeSegList をたどってゆき, 十分なサイズのオブジェクトを見つける必要がある. 最悪の場合, アロケーションのたびに freeSegList を最後までたどる必要があるため GC 処理の速度が低下する問題がある.

2.5 世代別 GC アルゴリズム

GC 速度の低下を回避して, より効率よく Mark and Sweep GC を行うために, 本研究では,Non-moving bitmap GC に加え, bitmap を利用した世代別 GC のアルゴリズムを拡張するものとした. 世代別 GC の設計は, 過去様々提案されている. 一般的にはオブジェクトを複数の世代 (領域) に分類し, 新世代 (young オブジェクト) を中心に GC を行う事で, GC 全体の効率を向上させる. これに対して, 世代を領域毎に分けるのではなく，bitmap ごとに分ける方式で世代別 GC を実装することで, より効率の良い世代別 GC を実現する手法が上野らにより提案された [4].

上野らの提案を参考に, 本研究でも世代別 Mark-and-Sweep GC を, Non-moving bitmap GC 上に設計し, 実装を行った. Non-moving bitmap GC は，トレースしたオブジェクトに対応する bitmap tree に bit がたっていない場合，そのオブジェクトにマーク処理を行う．そこで，初期化時にトレースして欲しくないオブジェクトには bit をたてることで，そのオブジェクトへのトレースを防ぐことができる．具体的に，Non-moving 世代別 GC の major GC では, GC 開始時に，全てのセグメントの bitmap tree の bit を 0 にする. しかし，minor GC の場合は，tenure オブジェクトの bit(oldBitmap) は保持することで，tenure オブジェクトのトレースを行わせないように設計した．(図 3).

次に, Mark 処理として，ルートセットから順にトレースを行い，トレースできるオブジェクトに対応する bitmap の bit をマークする．minor GC の場合，ルートセットからたどれるオブジェクト全てにマークがついた後，Remembered Set に登録されたオブジェクトをマークする.

ルートセット，Remembered set からたどれるオブジェクト全てにマーク処理が終わると，Sweep 処理に入る．Non-moving bitmap GC では deferred sweep を採用しているため，Sweep 処理では，メモリを走査して，解放を行うという処理はしない．代わりに，セグメントの livecount から，セグメントで利用できる block 数を計算し，空きの block があるなら，そのセグメントを freeSegList に連結する．Sweep 処理の後，Remembered Set のリストを初期化する.

3. 世代別 Incremental GC

前述のように, 我々は Non-moving 世代別 GC により, オーバーヘッドの少ない効率的な GC の設計および実装をスクリプト言語 konoha 上にて実現した. しかし，Non-moving 世代別 GC では, mutator の停止時間が不確定であるため, リアルタイム処理に適さない. 本研究では特に, 組込みシステムに向けたスクリプト言語を開発するにあたり, この点をさらに改善する必要があると考え, Incremental GC の設計を行った.

3.1 提案

課題に対応するため, 我々は Incremental GC によりリアルタイム性能を向上させる手法を検討した [1][5]. Incremental GC は, ユーザプログラムを実行するスレッド (mutator) の実行中に, GC を行うスレッド (collector) 処理を時間的に分散させる事で, 毎回の GC による mutator の停止時間を短縮するメリットがある. GC 処理に一定の作業量を設定し, 分割することで, ユーザプログラムの停止時間に上限を設けることが可能となるためである.

本研究では, Non-moving 世代別 GC によりオブジェクト指向型言語としてのスクリプト言語の特徴と, Incremental GC による mutator 実行時間の短縮を実現することで, オブジェクト移動のオーバーヘッドを最小限におさえつつ, 最大停止時間を削減することを目標とする.

本節では, 我々が実装した Non-moving Incremental GC の設計について述べ, その後, 応用として Non-moving 世代別 Incremental GC の設計について述べる．Non-moving bitmap GC を拡張して，世代別 GC と同じく 2 つの bitmap を用いた世代別 Incremental GC の設計を行なった内容について述べる.

3.2 Incremental GC

先に述べた konoha 言語の最初の実装である, Non-moving 世代別 GC では世代ごとに bitmap を用意した．しかし，

Non-moving Incremental GC ではマークとアロケート用に bitmap を 2 つ用意する．これは, mutator が segment の先頭の空オブジェクトを優先してアロケートするという仕様に対処したものである．Incremental GC の場合，collector が bitmap tree を 0 で初期化したあと，Mark 処理の途中で，mutator に操作が移る．このとき，bitmap tree を mutator, collector が共有していると，mutator は先頭のオブジェクトが空いていると誤認して，先頭にオブジェクトを新たにアロケートしてしまう．(Mark 処理中のため，先頭のオブジェクトが生きている可能性がある) そこで，それぞれの bitmap を用意することで，マークを中断してアロケーションが発生しても，bitmap の状態を保存できるようにする必要がある. この実現のため，segment の構造を (4) のように変更する．

$$Segment = (livecount, blocks, bitmaptree, markbitmaptree, RememberedSet) \quad (4)$$

collector は mark_bitmap にのみ書きこみを行い，mutator は bitmap にのみ書きこみを行う. GC 処理が終了すると，collector は mark_bitmap の内容を bitmap に反映させる．

3.2.1 アロケーション　アルゴリズム

本提案手法の特徴として，Incremental GC のオブジェクト生成時のオーバーヘッドの低下が挙げられる．まず，従来の Incremental GC のオブジェクト生成時のアルゴリズムは Incremental GC では，マーク処理とアロケート処理を交互に行うため，アロケート関数を (List. 1) のようにする必要がある．

アロケーションアルゴリズムを実現するこの関数では, GC の起動を確認し, GC が発生していない (GC_NONE_PHASE) ならばアロケートを行う．アロケートされたオブジェクトがしきい値を超えた場合，GC が起動し, collector は GC 初期化処理を行い，GC_MARK_PHASE に移る．collector が GC_MARK_PHASE か，GC_SWEEP_PHASE の場合，allocateCount がしきい値を超えるまで，アロケートを行う．allocateCount のしきい値は，GC 起動のしきい値と比べて小さくする．こうすることで, GC_NONE_PHASE の状態を長くし，プログラムの実行時間を短くする．allocateCount がしきい値を超えると，collector に処理が移る．従来の方法では，GC_MARK_PHASE か GC_SWEEP_PHASE の場合，オブジェクトをアロケートする度に，allocateCount の値以下かの判定を行う必要があり，これは Incremental GC のオーバーヘッドとなる．しかし，2.3 章で述べた通り，Non-moving bitmap GC はオブジェクトが端から順にアロケートされる．そこで，本提案手法では，GC 初期化時毎に Allocation Pointer がさしていたオブジェクトからしきい値分離れた bitmap を，次に Mark 処理が発生するトリガーとして設置する (List.2)．これは List. 2 の gc_init() 関数の際に，ALLOCATE_COUNT 分離れた bitmap にトリガーを設置することで，アロケートの際の条件分岐が減り，オーバーヘッドの低下になると我々は考えたためである．なお，ALLOCATE_COUNT の値を変化させるだけで，GC が起動するタイミングをずらせるため，GC が終了し，次の GC が発生するまでの間は ALLOCATE_COUNT を多めにとり，GC_NONE_PHASE を扱っている．

Listing 1　従来の Incremental アロケーション アルゴリズム

```
1  bitmap_allocate(mng, size):
2    obj = null
3    h = setSubHeap(mng, size)
4    if (GC_NONE_PHASE) :
5      obj = allocate(mng, h)
6    else : // GC_MARK_PHASE or GC_SWEEP_PHASE
7      if (allcateCount > MAX_ALLOCCOUNT) :
8        IncrementalGC(mng)
9        obj = allocate(mng, h)
10       allocateCount = 0
11     else :
12       obj = allocate(mng, h)
13       allocateCount += 1
```

Listing 2　提案する Incremental アロケーション アルゴリズム

```
1  gc_init(mng) :
2    foreach(h in SubHeaps) :
3      h.p.gcTriger.idx += ALLOCATE_COUNT / BITS
4
5  bitmap_allocate(mng, size) :
6    obj = null
7    h = setSubHeap(mng, size)
8    obj = allocate(mng, h)
9    if (h.p.gcTriger.idx == obj) :
10     IncrementalGC(mng)
```

3.2.2 GC アルゴリズム

Incremental GC では，最大停止時間の短縮のために，collector の処理と，mutator の処理を交互に行う．これは, mutator が segment の先頭の空オブジェクトを優先してアロケートするという仕様に対処したものである．Incremental GC の場合，collector が bitmap tree を 0 で初期化したあと，Mark 処理の途中で，mutator に操作が移る．このとき，bitmap tree を mutator, collector が共有していると，mutator は先頭のオブジェクトが空いていると誤認して，先頭にオブジェクトを新たにアロケートしてしまう．(Mark 処理中のため，先頭のオブジェクトが生きている可能性がある) そこで，マーク用の bitmap とは別に，アロケート用の bitmap を用意し，マークを行う bitmap とアロケートを行う bitmap を分ける．世代別 GC では世代ごとに bitmap tree を用意したのに対し，Incremental GC では mutator と collector に

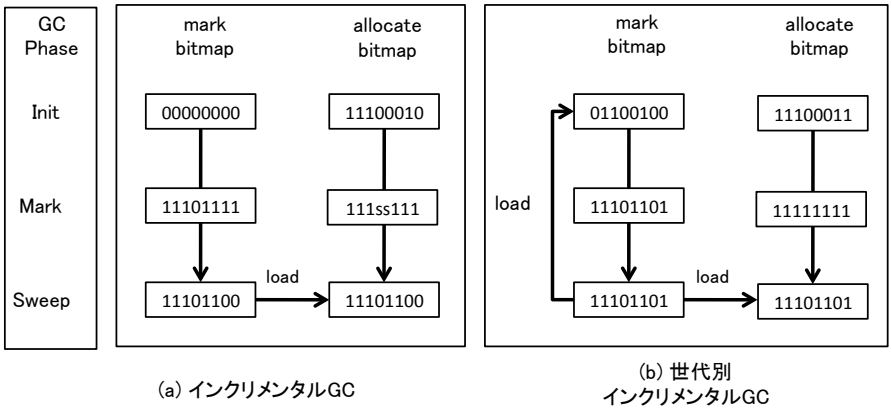

図 4 Incremental GC の bitmap tree の遷移図

bitmap tree を用意した．

Incremental GC の bitmap の遷移を図 4(a) に示した. List. 3 のようになっており，以下でアルゴリズムの説明を行う．

まず，マーク用の bitmap (mark_bitmap) は GC の初期化関数である gc_init_clearAllBitmap() にて全ての bit を 0 にして，マーク処理を行う．一方で，アロケート用の bitmap(alloc_bitmap) は引き続きアロケートに使用する．

マーク処理ではルートセットからたどれるオブジェクトをマークし，そこからたどれるオブジェクトを順次マークしていく．ルートセットからたどれる全てのオブジェクトへのマークが終了すると，write barrier によって登録されたオブジェクトをマークする．write barrier は snapshotGC のアルゴリズムに従い，GC 中に発生したポインタ付け替えでの，古い参照の登録を行う．

スイープ時には，空いているセグメントをサブヒープの freeSegList に繋げ，死んだオブジェクトの数をカウントする．(List. 3: 17 行目) このとき，全てのセグメントの空きを確認し，freeSegList に連結すると共に，copyMarkBitmap_to_AllocBitmap() で mark_bitmap を alloc_bitmap にコピーを行う. これにより，次回のアロケーションから, 死んでいたオブジェクトブロックを再利用することができる．死んだオブジェクトの数がセグメントのブロック数以下だった場合，そのサブヒープは満杯であると見なされ，ヒープの拡張が発生する．(List. 3: 25 行目)

3.3 世代別 Incremental GC

minor GC では，tenure オブジェクトの情報を load してから，GC を開始した．これは，新たにアロケートされた young オブジェクトに対応する bit のみを 0 で初期化するためである．しかし，mark_bitmap では新たにアロケートされたオブジェクト情報を書き込まないため，次の GC が起動するまで，mark_bitmap の状態は変化しない．GC が終了しても mark_bitmap の状態を保持しておくことで，young オブジェクトに対応する bit のみを 0 にしたと見なすことができる．そのため，Non-moving 世代別 Incremental GC では，世代別 GC と Incremental GC の bitmap 操作を組み合わせて行う．(図 4(b))

Listing 3 Non-moving Incremental GC アルゴリズム

```
1  gc_init_clearAllBitmap() :
2    foreach(h in SubHeaps) :
3      foreach(s in Segment) :
4        clearBitmap(s)
5        clearLivecount(s)
6
7  gc_init() :
8    gc_init_clearAllBitmap()
9
10 gc_sweep() :
11   foreach(h in SubHeaps) :
12     foreach(s in Segment) :
13       deadCount = segmentBlockCount − getLivecount(s)
14       if (deadCount > 0) :
15         pushFreeSegList(h, s)
16       copyMarkBitmap_to_AllocBitmap(s)
```

アルゴリズムは List.4 の通りである．minor GC の場合，マーク用の bitmap を GC 初期化時に 0 クリアする代わりに，以前のマークが終わった状態を load することで，tenure オブジェクトのマークを行わない．従って，gc_init() 関数の通り，初期化せずに，マーク処理を開始する．また，スイープ時は Incremental GC とほぼ同じだが，tenure_livecount の更新を行う．これは，tenure オブジェクトの数を更新することで，major GC を正しく起動させるためである．

4. 実装

前節にて設計した Non-moving Incremental GC と Non-moving 世代別 IncrementalGC を konoha に実装する際，konoha に合わせて実装を行った．実装では, write barrier と, リアルタイム処理に重要なセーフポイントについての改善点について述べる．特に write barrier は, collector と

Listing 4 Non-moving 世代別 Incremental GC アルゴリズム

```
1  gc_init()
2    if (isMajorGC) :
3      gc_init_clearAllBitmap()
4
5  gc_sweep() :
6    foreach(h in SubHeaps) :
7      foreach(s in Segment) :
8        deadCount = segmentBlockCount − getLivecount(s);
9        if (deadCount > 0) :
10         pushFreeSegList(h, s);
11       copyMarkBitmap_to_AllocBitmap(s);
12       copyTenureLivecount(s);
```

mutator が交互に動作するため Incremental GC において, 効率に重要な役割を果たす. 本節では，その詳細について述べる．

4.1 write barrier

konoha は世代別 GC の Write Barrier の際，参照元のオブジェクトが tenure かどうか，判断を行う．この操作は mutator にとってオーバーヘッドであり，コストを削減する必要がある．参照元のオブジェクトが tenure かどうか判断するためには，対応する bitmap を参照して，フラグがセットされているかどうが確認する方法が一番単純である．

しかし，オブジェクトから bitmap を参照するためには，blocks の先頭アドレスから，セグメント，bitmap と，ポインタの移動が発生する．ポインタの移動はキャッシュを汚す恐れがあり，空間的局所性が失われる．したがって，Write Barrier の度にポインタの移動が発生するとコストが高いため本提案手法では，konoha のオブジェクトが持つ，オブジェクトヘッダに tenure フラグを設置した．これにより，Write Barrier の際のオブジェクトの参照が無くなり，高速化が期待できる．

提案手法の Incremental GC における write barrier は，湯浅の snapshot GC[6] を実装した．snapshotGC は GC が起動した際にトレースできるオブジェクトのみを探索する GC であり，マーク中に新たに生成されたオブジェクトに関しては，無条件でマークを行う．そして，マーク中に参照元のオブジェクトがマーク済みで，元々参照先であったオブジェクトがマークされていない場合に rememberd set に元参照先のオブジェクト格納する．

4.2 GC セーフポイント

また，konoha の VM 命令には，GC セーフポイントのフラグが立っているかどうかを確認し，GC セーフであれば GC を行う命令がある．これはオブジェクトを新しく生成する直前に挿入される命令である．

通常は，オブジェクトを新しく生成し，初期化や代入処理が終了すると，mutator は生成されたオブジェクトを要求があったオブジェクトに参照させる．この初期化や，代入処理の途中で GC が発生した場合，生成されたがどこからも参照されていないオブジェクトになってしまい，GC はこれを死んでいるオブジェクトと判断し，解放してしまう．オブジェクト生成前に GC を行うようにすることで，GC セーフポイントで必ず GC が発生するような設計になっている．従って，空いているオブジェクトがなくなったら GC という実装では，GC セーフではないため，ある程度余裕を持って，GC セーフポイントのフラグを立てる．そこで，konoha に実装した Non-moving bitmap GC では以下のように GC 条件を定めた．

- 空いているセグメントがない
- セグメントに空いているブロックがしきい値以下である
- GC セーフポイントのフラグが立っていない

5. 性能評価

本節では，Non-moving 世代別 Incremental GC に加えて，Non-moving bitmap GC，Non-moving 世代別 GC，Non-moving Incremental GC に対して，実験を行い，比較，評価を行なった．

5.1 実験

本実験は，Intel(R) core i7(2.67 GHz) の CPU, 6GB memory，OS として Art-Linux(Ubuntu 10.04 32bit) という実験環境で行なった．また，実験に伴い，Art - Linux が提供している API を konoha に実装し，konoha でリアルタイムを提供するスクリプトを記述できるようにした．

リアルタイムシステムにおいて GC における処理の停止はデッドラインミスに大きな影響を与える．GC をリアルタイムシステムで用いる場合は，このデッドラインミスが発生しないように，デッドラインをあらかじめ設定しておく必要がある．あらかじめデッドラインの設定を行うためには，GC による停止時間の予想が付くこと，すなわち，GC による停止時間が常に一定であることが望まれる．

そこで，デッドラインミスを引き起こすかどうかを判定するために，GC による最大停止時間と，停止時間の偏り (jitter) として、今回は標準偏差で評価を行った．

実験に用いたベンチマークスクリプトは以下の通りである．

binary-trees 2 分木を生成するベンチマーク
movie 動画を再生するベンチマーク

binary-trees は本実装を評価するために作成した 2 分木を生成するベンチマークであり，1 回の木の生成にかかる時刻を計測した．生成にかかる時間をあらかじめ見積もり，デッドラインを 200[ms] とした．また，一般的なスクリプトとして，movie を用意した．movie は動画を流すべ

ベンチマーク [ms]	MSGC	GenGC	incGC	biGC	giGC
binary-trees	14.12	14.77	0.42	0.28	0.32
movie	0.84	0.11	0.073	0.075	0.071

表 1 最大停止時間

ベンチマーク [*E-9]	MSGC	GenGC	incGC	biGC	giGC
binary-trees	72.88	71.39	1.23	1.07	1.05
movie	0.012	0.042	0.81	0.25	0.30

表 2 停止時間の標準偏差

ンチマークであり，定期的に画像を更新するスクリプトのため，ソフトリアルタイムなベンチマークである．動画には，H.264 コーデックで幅 320，高さ 240[pixel] の動画を使用した．

5.2 評価

本節では，実験結果を元に，Mark and Sweep GC, 世代別 GC, Incremental GC, bitmap を用いた GC 起動を行う Incremental GC(bitmap Incremental GC とする), 世代別 Incremental GC の評価を行った．ベンチマーク結果は，それぞれ，MSGC, GenGC, incGC, biGC, giGC として表記する．

binary-trees においては，表 5.2 より，Incremental GC を実装した GC の最大停止時間が低かった．しかし，表 5.2 から標準偏差が最も低かったのは世代別 Incremental GC であった．リアルタイム処理においては、予測性が重要である。従って，停止時間の標準偏差が小さかった世代別 Incremental GC が有利であると言える．なお，世代別 Incremental GC の最大停止時間が snapshotGC と比較して，大きいという結果になった．これは，minor GC をインクリメンタルに実行した結果，飢餓状態に陥り，新たにセグメントをアロケートするための時間が余計にかかったためであると思われる．

また，movie のベンチマークにおいて，snapshot GC が有利な標準偏差を示したが，こちらは，動画を構成する，全ての画像ファイルがすぐにゴミとなるため，世代別の利点が生かせず，停止時間，標準偏差ともにその他の Incremental GC と差が出なかったのではないかと考えられる．

6. 関連研究

本節では，GC アルゴリズムの関連として，Non-moving bitmap GC をあげ，write barrier の実装として，snapshotGC を挙げる．

Non-moving bitmap GC はオブジェクトの移動をせずに，bitmap を用いてアロケーションを行う GC である [4]．マークに用いる bitmap をアロケーションにも利用し，GC によってオブジェクトの格納場所に隙間ができても，すぐにその隙間を埋めるようにアロケートが行われる．遅延スイープによって GC 時のヒープの一斉捜査によるスイープ処理をなくし，キャッシュが散らばることを抑えている．また，この bitmap を世代ごとに用意することで，世代別 GC を応用として実装し，スループットを向上させた．本論文では，スクリプト言語で求められている Mutator の停止時間について着目し，Non-moving 世代別 GC に Incremental GC を実装した．

湯浅らの snapshotGC[6] は GC が開始された時点でトレースできるオブジェクトを GC 対象とする GC である．従って，それ以降 (GC 中) に死んだオブジェクトは生きているオブジェクトとして扱われる．湯浅らはその write barrier の実装において，ポインタの参照が外れた場合に，その外れた参照先のオブジェクトをトレース対象にする．これによって，GC 開始時に生きていたオブジェクトは全てトレースできるようになる．また，GC 開始後に生成されたオブジェクトはマークされ，次回の GC から GC 対象となる．我々の実装との違いは，マークフェーズ，スイープフェーズに入った後に，ポインタの付け替え，オブジェクトの生成があった場合は，それらのオブジェクトを Remembered Set に入れた点である．これによって，GC 時間は増加してしまってでも，Mutator のオーバーヘッドを軽減させることを狙っている．

7. むすびに

本論文では，konoha において，Non-moving 世代別 Incremental GC の実装を行なった．実装にあたり，Non-moving bitmap GC の bitmap tree をマーク用と，アロケート用に複数用意することよって Incremental GC を実装した．その結果，Non-moving 世代別 Incremental GC によって，最大停止時間は MSGC の 2.66%，停止時間のジッタは標準偏差で 1.44%に抑えることができた．今後の方針としては，concurrent GC の実装を行い，さらに高速化を目指す予定である．

参考文献

[1] Richard Jones.
[2] Kimio Kuramitsu. Konoha: implementing a static scripting language with dynamic behaviors. In Workshop on Self-Sustaining Systems, S3 '10, pages 21–29, New York, NY, USA, 2010. ACM.
[3] Kimio Kuramitsu. Language design and implementation for scripting embedded applications. IPSJ Journal, 51(12):2185–2194, 2010-12-15.
[4] Katsuhiro Ueno, Atsushi Ohori, and Toshiaki Otomo. An efficient non-moving garbage collector for functional languages. SIGPLAN Not., 46:196–208, September 2011.
[5] Paul R. Wilson. Uniprocessor garbage collection techniques. In Proceedings of the International Workshop on Memory Management, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.
[6] T. Yuasa. Real-time garbage collection on general-purpose machines. J. Syst. Softw., 11:181–198, March 1990.