# UPMEM PIMのためのスクラッチパッドメモリをyoung世代とする世代別GC

森本 龍　鵜川 始陽

本研究では，UPMEM Processing-in-Memory(PIM) の DRAM プロセッサ (以下 DPU) の上で動く GC を提案する．UPMEM PIM はメモリモジュールの各 DRAM チップにプロセッサを実装したアクセラレータである．我々は Java のプログラムの一部の実行を UPMEM PIM にオフロードするシステムを開発している．本研究では DPU の持つスクラッチパッドメモリを世代別 GC の young 世代とすることで，マイナー GC 時の DRAM アクセスを減らし，高速にゴミを回収する GC を提案する．GC が何度か発生するプログラムにおいて，提案する GC とスクラッチパッドメモリを使用しない他の何種かの GC でそれぞれ処理時間と GC にかかる時間を比較し，スクラッチパッドメモリを使用しない世代別 GC よりも 1.5 倍から 3 倍の速さの時間で GC できることを確認した．

## 1　はじめに

Processing-in-Memory(PIM) は，メモリの近くにメモリプロセッサが配置された複数のノードからなるアーキテクチャである．プログラムをメモリプロセッサにオフロードし，データの存在するメモリプロセッサで計算を行うため，従来の CPU で計算を行うアーキテクチャと比べ，CPU とメモリの間でのデータの転送量を削減することが可能である．これにより CPU とメモリ間の転送量増大に伴うボトルネックを解消することが期待されている．

現在公式に唯一利用可能な PIM として，UPMEM PIM [8] が存在する．UPMEM PIM はメモリモジュールの各 DRAM チップにプロセッサ (以下 DPU) を実装したアクセラレータである．DPU の上でプログラムを動かす際には，CPU から DPU にプログラムをロードし，それを CPU から起動することで実行ができる．

我々は Java のプログラムの一部をこの UPMEM PIM にオフロードするシステムを開発している [5]. このシステムでは Java プログラムをバイナリにコンパイルし，それを DPU にオフロードして実行する．

本研究では，このシステムにおいて DPU の上で実行される Java 処理系の GC を開発する．高速な GC を実装する際には，DPU の持つスクラッチパッドメモリをどのようにして有効に使用するかが重要である．DPU はアクセス速度が速いが容量の小さいスクラッチパッドメモリと，容量が大きいがアクセス速度の遅いメインメモリを有している．そのためメインメモリのみにデータを置くようにすると GC を行う際にメインメモリへのアクセスが多発し，GC にかかる時間が増大することが予想される．

本研究では，young 世代とリメンバードセットをスクラッチパッドメモリに置く世代別 GC を提案する．young 世代をスクラッチパッドメモリ上に置くことで，マイナー GC 時にメインメモリへのアクセスを抑える．これによりマイナー GC にかかる時間を削減することが可能である．またマイナー GC を頻繁に行うことで，メインメモリに頻繁にアクセスするメジャー GC の頻度を抑える．

提案する GC を我々の開発している Java システムに対して実装した．実行の途中で何度も GC が発生

* Generational GC Using Scratchpad Memory as Young for UPMEM PIM.
This is an unrefereed paper. Copyrights belong to the Author(s).
Ryu Morimoto, Tomoharu Ugawa, 東京大学情報理工学研究科, Graduate School of Information Science and Technology, the University of Tokyo.

するようなプログラムにおいて，提案する GC とスクラッチパッドメモリを使用しない他の何種かの GC で処理時間や，GC にかかる時間を測定した．これにより，多くのオブジェクトが作られた直後にすぐ使用されなくなる状況において他の GC よりも高速に GC を行えることを確認した．またスクラッチパッドメモリを使用しない世代別 GC よりも 1.5 倍から 3 倍の速さの時間で GC できることを確認した．

## 2 背景

### 2.1 UPMEM

我々の研究では，実在する PIM アーキテクチャの計算機として UPMEM [8] を用いる．UPMEM では，各メモリモジュール内の DRAM チップにインメモリプロセッサーとして DRAM Processing Unit(DPU) が配置されている．

DPU は DRAM チップのメモリをメインメモリとして用い，それとは別にスクラッチパッドメモリを持っている．このメインメモリを MRAM，スクラッチパッドメモリを WRAM と呼び，それぞれ 64MB, 64KB の容量を持っている．DPU から WRAM のアクセスは直接行えるため高速だが，一方で DPU から MRAM へのアクセスには時間がかかる．これは MRAM へのアクセスは，データが WRAM に DMA 転送された後に行われ，直接アクセスができないためである．

WRAM に対する 4 バイトの読み書きは，11 サイクルかかる．これに対して MRAM へのアクセスは，$S$ バイトのデータの読み込みに約 $77 + 0.5S$ サイクル，書き込みに約 $61 + 0.5S$ サイクルかかると報告されている [4].

### 2.2 GC の実装における課題

MRAM へのアクセスは遅いので，GC は MRAM アクセスを抑える必要がある．Java オブジェクトを配置する先として MRAM と WRAM が考えられる．もし全てのオブジェクトを容量の大きな MRAM に配置することにすると，GC は生きているオブジェクトを探すために多数の MRAM アクセスを生じさせる．

例えばマークスイープ GC で，MRAM にあるオブジェクト $O$ が持つ参照を全て辿る操作は，次のように行われる．

1. $O$ のヘッダーから，クラステーブルのインデックスを読み出す
2. クラステーブルから，$O$ の参照フィールドがどこにあるかを読み出す
3. 各参照フィールド $F$ について，
   - 3.1. $F$ を読み出す
   - 3.2. 参照先のオブジェクトのヘッダーにあるマークビットを読み出し，マークされていなければ GC スタックに積む

ステップ 1 では 1 回，ステップ 2 では 2 回 MRAM にアクセスする．ステップ 3 では，参照フィールド 1 つにつきステップ 3.1 と 3.2 で 1 回ずつ MRAM にアクセスする．従って合計でオブジェクトが持つ参照の数を $R$ とした時，生きているオブジェクト 1 個あたり $3 + 2R$ 回 MRAM にアクセスする．さらにスイープでも MRAM にアクセスする．

## 3 提案手法

### 3.1 世代別 GC

本研究では young 世代とリメンバードセットを WRAM に，old 世代を MRAM に配置する世代別 GC を提案する．世代別 GC は，マイナー GC で多くのゴミを回収し，生き残ったものだけを old 領域に移動する．提案手法は young 世代とリメンバードセットを WRAM に配置することで，頻度の高いマイナー GC に伴う MRAM アクセスを削減する。多くの MRAM アクセスを伴い，時間のかかるメジャー GC の頻度は低い．

また提案手法では，GC 以外の処理時間の削減も期待できる．新しくオブジェクトを作る時には，割り当てたメモリにゼロを書き込んで初期化する．その後コンストラクタの実行でも新しいオブジェクトへの書き込みが生じる可能性がある．提案手法では，新しいオブジェクトは WRAM に作られるので，これらの書き込みが高速に行われる．

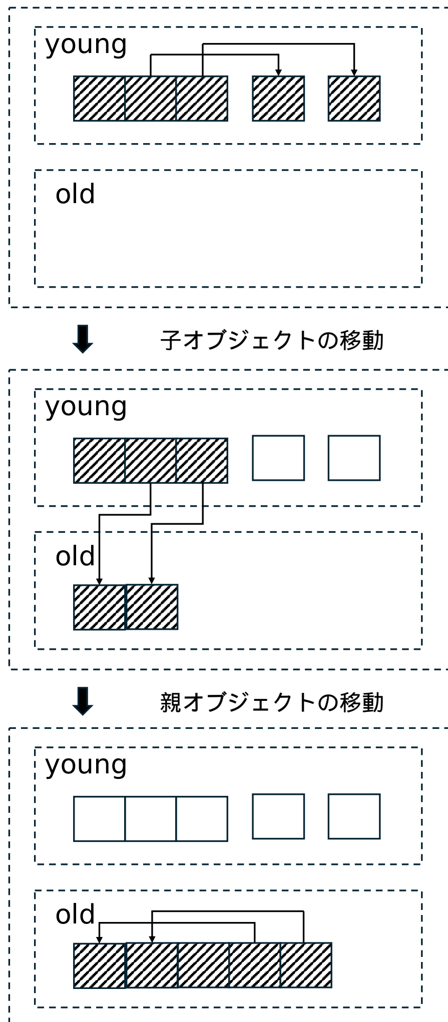

図 1: マイナー GC によるオブジェクトの移動

### 3.2 マイナー GC

新しく作られるオブジェクトは young 領域の空き領域の先頭に確保する．やがて young 領域に空きがなくなった際にはマイナー GC を行い，young 領域内の生きているオブジェクトを辿り，old 領域に移動する．この移動をオブジェクトの昇格という．

提案する GC では，深さ優先で参照を辿り，戻りがけ順にオブジェクトを移動することで，MRAM のアクセスを減らす．戻りがけ順でオブジェクトを移動すると，オブジェクトが WRAM 上にある間に子オブジェクトの移動が行われ，高速に子オブジェクトへのポインタを更新できる．図 1 はその様子である．まず

```
1 def copy(o)
2   oo = free_head
3   free_head += size(o)
4   o.mark = 1
5   o.forwarding = oo
6   cls = cls_tables[o.cls_index] // mram
7   ref_fields = cls.ref_fields   // mram
8   foreach f in ref_fields
9     p = o[f]
10    if in_young(p)
11      if p.mark == 1
12        pp = p.forwarding
13      else
14        pp = copy(p)
15      o[f]= pp
16  dma_copy(o, oo, size(o))      // mram
17  return oo
```

図 2: young 領域のオブジェクトを old 領域に移動する関数

子オブジェクトを移動する．その後親オブジェクトの参照を書き換える．この時親オブジェクトは WRAM 上にあるので，参照の書き換えは WRAM への書き込みになる．最後に親オブジェクトを移動する．代表的なコピー GC である Cheney のコピー GC [2] では，親オブジェクトをコピーした後に子オブジェクトをコピーする．このような場合には MRAM 上で子オブジェクトへのポインタを更新することになり，多くの MRAM アクセスが発生する．

図 2 にオブジェクトを young 領域から old 領域にコピーする関数 `copy` の疑似コードを示す．引数に渡されるオブジェクト `o` が，young 領域から old 領域に移動される．空いている old 領域の先頭である `free_head` を移動先とする．移動先が決まると，`o` のオブジェクトヘッダーのマークビットである `mark` に 1 を立てる．また同様にオブジェクトヘッダー内にある `forwarding` に移動先を書き込む．11，12 行目ではこの 2 つを使用することで，重複してオブジェクトを辿ることを避ける．次に `o` のもつ参照を辿る．MRAM にあるクラステーブルへアクセスを行い，そこから参照フィールド `ref_fields` を得る．その後はそれぞれの参照フィールドが指すオブジェクトについて，深さ優先でオブジェクトを辿る．深さ優先でオブジェクトを辿り，全ての参照フィールド `o[f]` を書き換えた後に，オブジェクトを移動することで，一度の DMA 転送のみの MRAM アクセスに抑えること

ができる．生きているオブジェクトを辿る際に，2.2 節で説明したマークスイープ GC では $3 + 2R$ 回の MRAM アクセスであった．この移動方法では `mram` とコメントがある箇所が MRAM アクセスとなっており，3 回の MRAM アクセスにまで削減できている．我々の実装では再帰関数によってオブジェクトを移動するためスタックオーバーフローが生じる可能性があるが、この問題については本論文では議論しない．

### 3.3 リメンバードセット

世代別 GC では old 領域のオブジェクトから young 領域のオブジェクトへの参照を辿るために，リメンバードセットを用いる．提案手法ではこのリメンバードセットを WRAM 上に置き，old 世代オブジェクト中の young 世代への参照フィールドへのポインタを格納する．マイナー GC が終了すると young 領域の生きているオブジェクトは全て old 領域に移動されるので，リメンバードセットの中身は空にする．

提案手法ではリメンバードセットに重複したものが登録されるのを許す．これは，例えばリメンバードセットに old 領域のオブジェクトを登録し，そのオブジェクトにマークを付けるようにすれば重複した登録は避けられるが，これが MRAM アクセスとなるためである．そのため参照をよく書き換えるプログラムでは，old 世代オブジェクトのフィールドを young 世代オブジェクトに書き換える度に，リメンバードセットに格納される．これによりリメンバードセットが満杯になる可能性が高くなる．リメンバードセットが満杯になった時は，マイナー GC をすることでリメンバードセットの中を空にすることで対応する．

### 3.4 メジャー GC

提案手法では old 領域を半分に分けて片方のみを使用し，その中にオブジェクトを順に詰めて配置していく．半分に分けたヒープのうち，現在使用している方を from 空間，もう一方を to 空間と呼ぶ．from 空間が満杯になるとメジャー GC を行う．その時の様子を図 3 に示す．メジャー GC では，young 領域と from 空間で生きているオブジェクトを全て to 空間に移動する．

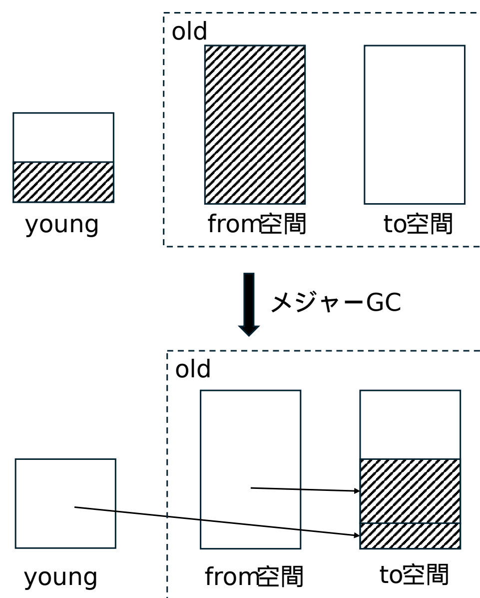

図 3: メジャー GC におけるオブジェクトの移動

移動する際にはオブジェクトを辿るのだが，この時の移動方法は図 2 の疑似コードとほぼ同じで，10 行目における young 領域にあるオブジェクトかどうかのチェックは行わない．全ての生きているオブジェクトの移動を終えると，メジャー GC が完了する．メジャー GC を行うと，young 領域内で生きているオブジェクトは全て old 領域に移動され空になるので，リメンバードセットの中身も空にする．

メジャー GC はマイナー GC に比べて，長い時間がかかることが予想される．これは young 領域上のオブジェクトを辿るのに比べて，old 領域上のオブジェクトを辿る際には MRAM へのアクセスが増えてしまうためである．メジャー GC で old 領域にあるオブジェクトを辿って移動する際は，図 2 の疑似コードにおいて移動されるオブジェクト `o` は MRAM にある．そのため 4，5 行目のヘッダーへの書き込みが MRAM アクセスとなる．また参照フィールドを辿る際には，9 行目の参照フィールドの読み込みと 15 行目の参照フィールドの書き換えが MRAM アクセスとなる．これらにより参照フィールドの数を $R$ とすれば，$2+2R$ 回 MRAM へのアクセスが増えてしまう．

このアルゴリズムでは，old 領域にオブジェクトを移動する際に，`free_head` を見るだけで移動場所がわ

かる．しかしメジャー GC 時にオブジェクトを移動するため，移動を行わないマークスイープ GC などと比べると，生きているオブジェクトの分だけ MRAM に対しての書き込みが生じ，その分時間がかかる可能性がある．

## 4 実験

本章では，提案する GC は他の GC よりも高速であるのか，また高速である場合に，その理由は young 世代を WRAM に置いたからなのかという疑問に答えるための実験を行った．

実験は以下の構成の UPMEM システムで行った．

- UPMEM PIM モジュール (バージョン v1A)
  - DPU (350 MHz)
- UPMEM SDK: 2023.2.0

DPU は 1 つのみを使用した．また DPU は最大で 24 個のスレッドをサポートするが，1 スレッドのみを用いた．

### 4.1 実験方法

GC が何度も発生するようなプログラムを実行し，GC 以外の実行時間と GC に要する時間を測定する．プログラムには 4.2 節で説明するようなものを用いる．

DPU 上の Java 処理系に提案する GC を実装し，比較対象として WRAM を使用しない世代別 GC(GEN)，マークスイープ GC(MS)，コピー GC(CP) を用意した．それぞれの GC の構成は以下の通りである．

- 提案手法
  - WRAM に 28KB の young 領域，4KB のリメンバードセット
  - MRAM に 1MB の old 領域
- GEN
  - MRAM に 28KB の young 領域，4KB のリメンバードセット
  - 992KB の old 領域
- MS
  - MRAM に 1MB のヒープ領域
- CP
  - MRAM に 1MB のヒープ領域
  - アルゴリズムは，提案手法における old 領域の管理と同様

### 4.2 実験プログラム

我々は以下の 2 つの量を制御できるベンチマークプログラムを作成した．

1. マイナー GC で昇格するオブジェクトの割合
2. old 領域内で生存しているオブジェクトの量

具体的には，木構造のデータに要素を追加したり，削除したりするプログラムを作成した．木に要素を追加する際に，余分に $K$ 個のオブジェクトを作り，それらは木に繋がずゴミとする．木の要素数は 600 を超えないようにし，超えると木の半分の要素を削除する．$K$ の値により作ったオブジェクトのうち何%が生きたオブジェクトとなるかを変更する．これにより昇格するオブジェクトの割合を制御する．以下では

$$R = \frac{1}{K+1}$$

とする．木の半分の要素を削除するイベントが起きなければ，$R$ が昇格率となる．このイベントは我々の実験では，最大でもマイナー GC 12 回に 1 回しか起きなかったので，以下では $R$ で昇格率を近似する．またこの木とは別に，プログラムの実行を通して変更されない木を作る．この木は常に old 領域に存在する．この木を構成するオブジェクトのサイズの合計を $S$ とする．old 領域には，この木に加えて，要素を追加，削除する木の一部も存在する．その大きさは，最大で 30KB 程度である．

### 4.3 実験結果

#### 4.3.1 old 領域への昇格率と世代別 GC の性能

マイナー GC 時に young 領域から old 領域に昇格するオブジェクトが少なければ，世代別 GC と提案手法が他の GC よりも高速となる結果が得られた．図 4 から図 6 にそれぞれの $S$ において GC に要した時間を示す．$R$ が 10%以下であれば，全ての $S$ で世代別 GC と提案手法が他の GC よりも高速に GC を行えていることがわかる．

しかし $R$ が大きくなると，世代別 GC や提案手法よりもマークスイープ GC の方が高速になっている．これの原因の 1 つとして，$R$ が大きいと old 領域に昇格

表 1: 各 $S$ と $R$ における GC の回数とその平均サイクル数

| GC の種類 | GC の回数 $S=50KB$ $R=25\%$ | 平均サイクル数 $S=50KB$ $R=25\%$ | GC の回数 $S=50KB$ $R=50\%$ | 平均サイクル数 $S=50KB$ $R=50\%$ | GC の回数 $S=250KB$ $R=25\%$ | 平均サイクル数 $S=250KB$ $R=25\%$ |
|---|---|---|---|---|---|---|
| CP | 16 | $1.60\times10^7$ | 15 | $1.53\times10^7$ | 34 | $5.92\times10^7$ |
| GEN(メジャー GC) | 4 | $7.64\times10^6$ | 8 | $8.24\times10^6$ | 10 | $2.96\times10^7$ |
| GEN(マイナー GC) | 231 | $7.58\times10^5$ | 227 | $1.48\times10^6$ | 225 | $7.68\times10^5$ |
| Proposed(メジャー GC) | 4 | $6.27\times10^6$ | 8 | $6.36\times10^6$ | 9 | $2.54\times10^7$ |
| Proposed(マイナー GC) | 232 | $2.67\times10^5$ | 228 | $5.27\times10^5$ | 227 | $2.73\times10^5$ |
| MS | 8 | $1.64\times10^7$ | 8 | $1.67\times10^7$ | 11 | $2.13\times10^7$ |

後すぐゴミになるオブジェクトが増え，メジャー GC の回数が増えていることが考えられる．表 1 に，各昇格率と生存オブジェクト量における，GC の回数と GC 1 回あたりの平均サイクル数を示す．$S=50KB$，$R=25\%$ と $S=50KB$，$R=50\%$ を比べると，$R$ が 2 倍になると世代別 GC と提案手法でメジャー GC が起きた回数も 2 倍となっている．提案手法において，オブジェクトは 1 度のマイナー GC を生き残れば，すぐに old 領域に移動される．これにより $R$ が大きいときに，old 領域がすぐに満杯となってしまい，メジャー GC が頻繁に行われてしまう．これの対策として，young 世代を複数設け，オブジェクトをエイジングするという手法 [6,7] が考えられる．

#### 4.3.2 世代別 GC 間の比較

WRAM を使用しない世代別 GC と提案手法を比べると，young 領域とリメンバードセットを WRAM に置いたことにより，提案手法の方が短い時間で GC できていることがわかる．提案手法と世代別 GC では，同じ世代別 GC のアルゴリズムを用いており，違いは提案手法では WRAM に young 領域とリメンバードセットがあることである．図 4 から図 6 では，昇格率や生存オブジェクト量によらず，提案手法において GC に要する時間は，世代別 GC よりも 1.5 倍から 3 倍程度速くなっている．表 1 では，昇格率や生存オブジェクト量によらず，提案手法のマイナー GC は，世代別 GC のマイナー GC よりも約 3 倍速くなっている．またメジャー GC にかかるサイクル数も 10%から 20%削減されている．これはメジャー GC 時には young 領域も探索するが，これが WRAM 上にあるためと考えられる．

#### 4.3.3 オブジェクトの移動と GC の性能

MRAM のみを用いるコピー GC とマークスイープ GC を比べると， マークスイープ GC の方が高速であることがわかった．図 4 から図 6 ではどのケースでも，コピー GC に要する時間の方がマークスイープ GC のものより長くなっている．またその差は $S$ が大きくなるほど顕著である．また表 1 において，$S=50KB$，$R=25\%$ と $S=250KB$，$R=25\%$ を比べると，$S$ が大きくなるとコピー GC にかかるサイクル数がマークスイープ GC に比べて極端に大きくなっている．これの原因の 1 つとして，コピー GC では MRAM の上でオブジェクトの移動を行うためであると考えられる．$S$ が大きいとより多くのオブジェクトを MRAM 上で移動し，その度に MRAM アクセスが生じる．またヒープを 2 分割して使用しているため，マークスイープ GC よりも GC の回数が多くなるのも原因の 1 つである．

提案手法においても，GC 時にオブジェクトの移動を行っている．図 4 と図 5 において，提案手法は $R=25\%$ の時にマークスイープ GC よりも高速に GC できている．しかし図 6 においては，$R=25\%$ の時にマークスイープ GC の方が高速となっている．$S$ が大きくなると，マークスイープ GC に対する提案手法の優位性が弱くなっていることがわかる．これの対策として，old 領域をオブジェクトを移動しないアルゴリズムで管理する GC が考えられる．

### 4.4 GC 以外の処理時間の比較

young 領域を WRAM に置くことにより，GC 以外の計算時間が 1%から 6%削減されるのを確認した．図 7 は各生存オブジェクト量における，世代別 GC

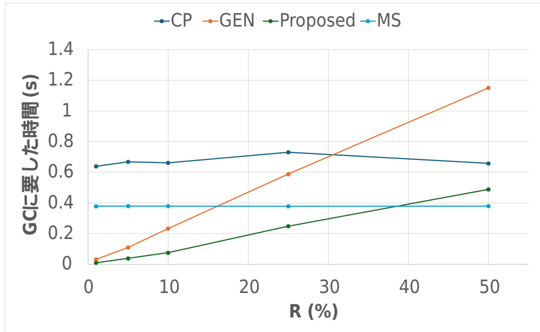


図 4: $S = 50KB$ における GC の実行時間

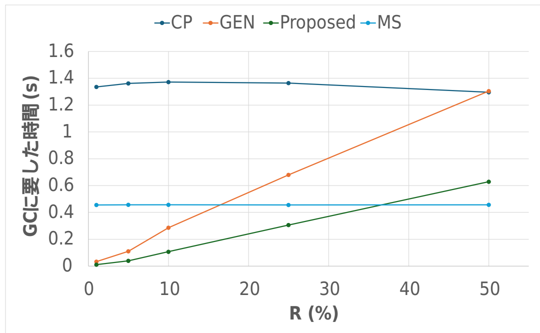


図 5: $S = 100KB$ における GC の実行時間

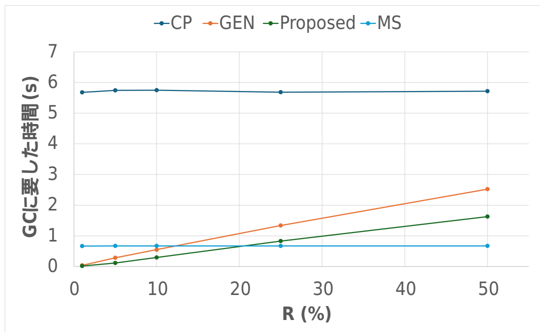


図 6: $S = 250KB$ における GC の実行時間

に対する提案手法の GC 以外の計算時間の比率である．どの生存オブジェクト量や昇格率においても，提案手法は世代別 GC よりも GC 以外での計算時間が削減されているのがわかる．これは提案手法において，young 領域にあるオブジェクトに対するアクセスが WRAM 上で行われるためと考えられる．提案手法においては，新しくオブジェクトを作成した際に，ゼロ初期化やコンストラクタにおける書き込みが WRAM 上で行われる．またプログラム中で木にオブジェクトを繋ぐ際に，木の要素に対する読み込みや書き込みが生じる．木の要素は young 領域か old

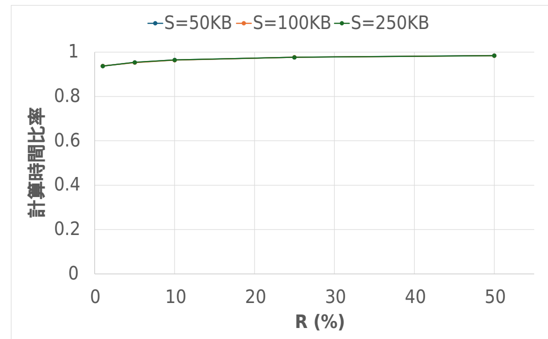


図 7: 提案手法の GEN に対する GC 以外の計算比率

領域のどちらかにあり，young 領域にある場合には WRAM 上でアクセスが行われる．これらが寄与し，GC 以外の時間が削減されたと考えられる．

## 5 関連研究

世代別 GC において，メモリアクセス時間を課題とした研究はこれまで行われている．近年プロセッサの処理能力が大幅に向上しているのに対し，メモリのアクセス速度の向上は限定的であり，メモリアクセスがボトルネックとなっている．またキャッシュは年々大きくなっており，これらに伴いキャッシュミスのコストがとても大きくなっており，Java アプリケーションにおいては，実行時間の 45%がメインメモリの待ち時間となる可能性があると報告されている [1]．キャッシュからメインメモリに書き戻す際には余計なメインメモリへの書き込みが生じている．Java においてほとんどのオブジェクトは 32 バイト以下であり，これは 1 つのキャッシュラインに収まる．またオブジェクトによっては 2 つのキャッシュラインに跨って存在するため，対象とするオブジェクト以外の領域も同時に書き込まれてしまう．また世代別 GC においては，キャッシュ上にあるゴミとなったオブジェクトもメインメモリへ書き戻されてしまう．これらを解決するために，世代別 GC においてキャッシュを young 世代として用いる研究が行われている．Yau ら [9] では並行 GC において 90%のオブジェクトをキャッシュに直接確保し，それらの 68%をキャッシュから回収できたと報告されている．この手法では死んだオブジェクトをキャッシュ上からメインメモリに書き戻す必要が

ない．

Chong ら [3] ではキャッシュの代わりにスクラッチパッドメモリ (SPM) に young 世代を置くシステムが提案されている．キャッシュと比べて SPM では，キャッシュミスがなく，確実にデータが存在するためデータへのアクセスを 1 サイクルで行うことができる．また SPM 上でメモリ空間を分けることが可能であり，複数のメモリ領域を設けて，それぞれで GC を行うことができる．これらの利点を利用し，[3] では young 世代オブジェクトと長命でよくアクセスされるオブジェクト (hot-mature オブジェクト) 領域を SPM に配置している．我々の手法でも，hot-mature オブジェクトを WRAM 上に置くことで，より高速化できる見込みがある．

## 6 まとめと今後の課題

本研究では，DPU 上で動く Java システムに対して GC を開発し，その性能を評価した．DPU の持つ高速にアクセスが可能なスクラッチパッドメモリに young 領域を配置し，そこで頻繁にマイナー GC を行い，アクセス速度の遅いメインメモリへのアクセスが多発するメジャー GC の頻度を減らした．これによって，スクラッチパッドメモリを使用しない世代別 GC よりも高速に GC を行えることを示した．また世代別 GC が得意とする，多くのオブジェクトが作られた直後にすぐ使用されなくなる状況において，他の何種かの GC よりも高速に GC を行えることを示した．また young 領域に高速にアクセスできることによる，GC 以外の処理の高速化も確認した．

今後の課題として，エイジングを用いて短命なオブジェクトの昇格を抑制することや，オブジェクトを移動しないアルゴリズムで old 領域を管理すること，hot-mature オブジェクトを WRAM に配置することが挙げられる．

**謝辞** 本研究の一部は，JSPS 科研費 23K24822 の助成を受けたものです．

## 参 考 文 献

[1] Adl-Tabatabai, A.-R., Hudson, R. L., Serrano, M. J., and Subramoney, S.: Prefetch injection based on hardware monitoring and object metadata, *ACM SIGPLAN Notices*, Vol. 39, No. 6(2004), pp. 267–276.

[2] Cheney, C. J.: A nonrecursive list compacting algorithm, *Communications of the ACM*, Vol. 13, No. 11(1970), pp. 677–678.

[3] Chong, K. F., Ho, C. Y., and Fong, A. S.: Pretenuring in java by object lifetime and reference density using Scratch-Pad memory, *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, IEEE, 2007, pp. 205–212.

[4] Gómez-Luna, J., Hajj, I. E., Fernandez, I., Giannoula, C., Oliveira, G. F., and Mutlu, O.: Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture, *arXiv preprint arXiv:2105.03814*, (2021).

[5] Huang, W. and Ugawa, T.: An Object-Oriented Programming Model for Processing-in-Memory Computing in Java, *Proceedings of the 40th JSSST Annual Conference*, 2023.

[6] Lieberman, H. and Hewitt, C.: A real-time garbage collector based on the lifetimes of objects, *Communications of the ACM*, Vol. 26, No. 6(1983), pp. 419–429.

[7] Ungar, D.: Generation scavenging: A non-disruptive high performance storage reclamation algorithm, *ACM SIGPLAN Notices*, Vol. 19, No. 5(1984), pp. 157–167.

[8] UPMEM: UPMEM, 2024. https://www.upmem.com/technology/.

[9] Yau, C. H., Tan, Y. Y., Fong, A. S., and Yu, W. S.: Hardware concurrent garbage collection for short-lived objects in mobile java devices, *Embedded and Ubiquitous Computing–EUC 2005: International Conference EUC 2005, Nagasaki, Japan, December 6-9, 2005. Proceedings*, Springer, 2005, pp. 47–56.