

# ゴミ集めの機能を応用した 不揮発性メインメモリへの書き込み抑制手法の予備的評価

中川 岳<sup>1,a)</sup> 追川 修一<sup>2,b)</sup>

受付日 2013年4月12日, 採録日 2013年9月19日

**概要:** プロセッサからバイトアクセス可能な不揮発性メモリ (NVM) の実用化研究が進み, 主記憶としての利用が検討されている. 主記憶への NVM の採用は, 計算機の消費電力削減や, より柔軟なシステム中断と高速復帰が可能になるなど, メリットが大きい. 現在開発されている NVM には, 書き込みボトルネックが大きい, 書き込み回数に制限があるなどの問題がある. その解消のために, DRAM と NVM を組み合わせたハイブリッドメモリアーキテクチャが提案されている. ハイブリッドメモリアーキテクチャの活用には, プログラムのデータ書き込み頻度を収集し, それに基づいたデータ配置先の振り分けを行う必要がある. 振り分けは, ハードウェアレベル, オペレーティングシステムのレベルで実現する方法が提案されている. しかしながら, 下位レイヤでの手法は, 幅広いプログラムを対象にした書き込み頻度の収集と振り分けが行える一方, データの意味に基づいた振り分けを行うことができない. 本論文では, 言語処理系が持つライトバリア処理に着目し, 言語処理系のレベルでデータ配置先の振り分けを行うことで, NVM への書き込みを抑制する手法を提案する. また, 提案手法を実装した言語処理系を用いて予備的評価を行い, 提案手法の効果や問題点を検討する.

キーワード: 不揮発性メモリ, 省電力, 言語処理系

## A Preliminary Evaluation of a Write Reduction Method for Non-volatile Main Memory Utilizing Garbage Collector

GAKU NAKAGAWA<sup>1,a)</sup> SHUICHI OIKAWA<sup>2,b)</sup>

Received: April 12, 2013, Accepted: September 19, 2013

**Abstract:** There are many researches and developments on byte accessible non-volatile (NV) memory. Byte accessible NV memory can be used as not only storage but also main memory. Non-volatile main memory has two significant merits: reducing power consumption of main memory, flexible system standby and fast resume. However, NV memory has bottlenecks in writing and limitation on the number of writing. To overcome this problem, there are some researches on hybrid memory architectures, which combine NVM and DRAM. On hybrid memory architectures, it needs to collect program characteristics of writing data and allocate either DRAM or NVM based on that characteristics. Previous works proposed to collect and allocate at hardware level or operating system level. At hardware or OS level, memory allocators can collect writing information of wide processes. However, they do not know contents of written data; thus they cannot allocate memory based on data characteristics. To solve this problem, we propose a method to obtain characteristics of writing and assign either DRAM or NVM based on that information. The method applies write barrier of language runtimes to collect writing characteristics.

**Keywords:** non-volatile memory, low power consuming, language runtime

<sup>1</sup> 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻

Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan

<sup>2</sup> 筑波大学システム情報系情報工学域

Division of Information Engineering, Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan

a) gnakagaw@cs.tsukuba.ac.jp

b) shui@cs.tsukuba.ac.jp

### 1. はじめに

プロセッサがバイトアクセス可能な不揮発性メモリ (Non Volatile Memory: NVM) の実用化研究がさかんである. 不揮発性メモリとしては NAND Flash や NOR Flash が広く普及しているが, バイト単位での書き込みができないことや書き込みレイテンシなどの制約から, これまで

主記憶として用いられることはなかった。一方で、PCM, ReRAM, MRAM などの読み書きともにバイトアクセスが可能であり、そのため主記憶に利用可能な NVM が開発されつつある [1], [2]。主記憶の DRAM を NVM で置き換えることで、主記憶のリフレッシュ処理の省略やシステムの高速起動などが実現でき、消費電力の削減や柔軟なシステムの利用が可能となる。

この NVM を主記憶として利用するにあたっての課題が 2 つある。1 つは書き込み速度が読み込み速度に対して低速であること、2 つめは NVM の種類によっては書き込み回数に制限がある点である。これらの課題は、実用化研究がさかんである PCM に顕著であり、それを克服するための研究が進められてきた [3], [4]。

NVM のデメリットを克服するための研究のなかでも、主記憶の DRAM を NVM で完全に置き換えるのではなく、DRAM と NVM を組み合わせて主記憶を構成することでこの問題を克服する研究が行われている。DRAM と NVM のハイブリッド構成（以下、ハイブリッドメモリアーキテクチャと表記する）においては、その目的から、DRAM には書き込み頻度の高いデータを、そうでないデータは NVM に配置することが望ましい。これを実現するためには、メモリへの書き込みに関する特性を収集し、それに基づいたデータ配置先を選択する必要がある。このようにあるデータに対して、その書き込み回数やデータの種類の特性に基づいて、ハイブリッドメモリアーキテクチャ上の NVM, DRAM 間で適切な配置先を決定、または再決定することを「データの振り分け」もしくは単に「振り分け」と定義する。ハイブリッドメモリアーキテクチャにおいて、データの記録先を決定する手法としては、オペレーティングシステム (OS) のメモリ管理機構レベルで行う手法が研究されている [5]。しかしながら、下位レベルでは書き込まれるデータが何を表現しているのかを詳細に知ることができない。プロセスのセクション情報やメモリにロードされたファイルの属性を基に、データの書き込み頻度を推定する研究 [6] もあるが、得られる情報の粒度は大きく、柔軟性のある振り分けはできない。このように、下位レイヤでは、書き込まれたデータの意味 (セマンティクス) に基づいて書き込み頻度を収集し、振り分けを行うことが困難である。

一方で、プログラミング言語の処理系レベルでは、セマンティクスを把握することができる。つまり、言語処理系レベルでは、下位レベルの振り分けよりも、より細やかな振り分けを行うことが可能になる。言語処理系レベルでプログラムが行うメモリ書き込みを把握するためには、処理系に大規模な修正が必要である。しかしながら、一部の言語処理系が持つライトバリア機構を応用することで、比較的容易に実現することができる。

以上をふまえ本論文では、ライトバリア機構を応用し、

プログラムのメモリ書き込み頻度の収集を行い、その特性を基に NVM への書き込みを抑制する手法を提案する。また、提案手法を実際の言語処理系に実装し、予備的な評価実験を行い、提案手法の効果とオーバーヘッドを評価する。なお、これ以降、NVM と表記した場合は、バイトアクセスが可能であり、主記憶として利用可能である PCM, MRAM, ReRAM などを目指すものとする。

本論文の構成は次のとおりである。2 章では研究の背景を述べる。3 章では提案手法を述べ、4 章ではその実装について述べる。5 章では提案手法による書き込み頻度の収集と結果について述べる。6 章では、5 章での結果に基づいた NVM, DRAM 間での振り分けシミュレーションとその結果について述べる。7 章では関連研究について述べ、8 章でまとめと今後の課題について述べる。

## 2. 背景

本章では、研究の背景となるハイブリッドメモリアーキテクチャについて述べる。また、その有効活用のために、メモリ上のデータの書き込み頻度収集の必要性について述べる。

### 2.1 ハイブリッドメモリアーキテクチャ

NVM は主記憶を置き換え可能な特性を持つ一方、書き込みレイテンシや耐久性の面で制約もある。そこで、NVM と DRAM をハイブリッド構成 (ハイブリッドメモリアーキテクチャ) にすることで、DRAM の特性で NVM の制約を補い、この問題を解決することが模索されている。図 1 にハイブリッドメモリアーキテクチャの構成とそのねらいについて示す。DRAM には高速な書き込みアクセスが可能である、書き込み回数が無制限であるという特性がある。この 2 つを組み合わせることで、NVM の書き込みボトルネックが懸念される場面では DRAM を使うことで、ボト

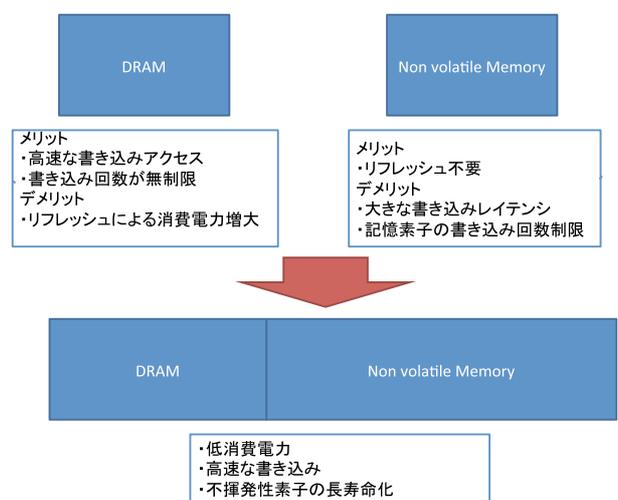


図 1 ハイブリッドメモリアーキテクチャのねらい  
Fig. 1 Aims of hybrid memory architecture.

ルネックを回避することができる。主記憶の一部に消費電力の大きな DRAM を使用しているが、従来の主記憶よりもその量を減らすことが可能であるため、主記憶全体としては消費電力の削減につながる。

## 2.2 書き込み頻度収集の必要性

前節で述べたハイブリッドメモリアーキテクチャを、その狙いどおりに利用するためには、書き込み頻度が多いデータについては DRAM に配置し、書き込み頻度が低いデータについては NVM に配置する必要がある。これを実現するにはメモリ上のデータに対してどの程度の頻度で書き込みが行われるかの情報が必要である。また、振り分けの基準を定める必要がある。そのためには、実際に動作しているプログラムからデータ書き込み頻度を収集する必要がある。プログラムのデータ書き込み頻度と振り分けの基準は、シミュレーションやプログラムの静的解析で推定することも可能ではあるが、現行の計算機システムで実行されるワークロードは多様であること、特性の違うワークロードを組み合わせたタスクが想定されることから、静的に推定した特性や基準だけでは不十分である。そのため、プログラムの挙動に合わせて動的に基準を定め、必要に応じて変更することが必要である。

また、データそのものでなく対象となったデータが持つ意味(セマンティクス)ごとの書き込み頻度も NVM のボトルネック回避には重要である。なぜなら、セマンティクスによって、書き込み頻度に傾向があることが予想されるからである。たとえば、リンクリストの先頭ポインタを表すデータは頻繁に書き換えられる可能性がある一方、文字列を表すデータは読み込みの対象となっても、書き換えの対象とはなりにくい。このようなセマンティクスごとの書き込み頻度が分かれば、新たに生成されて書き込み頻度が不明なデータについても、同種のセマンティクスから書き込み頻度を予測することが可能になる。これはセマンティクスを持つデータの振り分けの効率化につながる。このためにも、実行時の動的な書き込み頻度収集が必要である。

## 2.3 下位レベルでの書き込み頻度の収集

下位レベルでメモリ書き込みを検知するためには、いくつかの手法がある。ここでは2つの方法について議論する。1つめは OS のメモリ保護機構を応用する方法である。メモリ管理ユニット (MMU) を持つプロセッサでは、OS が指定した特定の領域へのメモリアクセスが起こった際に、例外を発生させることができる。この機能を利用すれば、メモリ上の特定の領域への書き込みを検知することが可能である。しかしながら、プロセッサの例外処理はプロセッサのコンテキストスイッチをとめない、コストが大きい。そのため、例外処理の増加によりシステム全体の性能が低下する可能性がある。

2つめの方法は、OS のページング機構を応用することである。メモリ管理にページング方式を採用している OS ではページアウトを実現するために、書き込みが起こったメモリページに印をつけることが一般的である。その印の中には Dirty ビットや Write ビットと呼ばれるものがある。この Dirty ビットや Write ビットをポーリングすることで、OS レベルからどのページに対して書き込みが起こったのかを把握することができる。この方法では、ポーリング間隔を調整すれば OS にかかる負荷を一定にできるため、割込みベースであったメモリ保護機構を利用する方式に対して、動作性能の低下をコントロールすることができる。しかしながらこの方式では、観察間隔の間に起こった複数回の書き込みは検知することができない。またどちらの方式も、収集できる書き込み頻度はページ単位になるので、振り分けの柔軟性を損なってしまう。いずれの方法にしても、データの書き込みは検知できても、データの詳細なセマンティクスを把握することはできず、それに基づいた振り分けを行うことはできない。

## 3. 提案手法

本章では、書き込み頻度収集の手段として、オブジェクト指向言語の言語処理系を利用した手法を提案する。また、それにより得られた書き込み頻度を基にメモリ書き込みの振り分けを行う手法を提案する。

### 3.1 ライトバリアの応用

オブジェクト指向言語は、プログラムが使用するデータをオブジェクト単位でメモリに記録する。そのため、プログラムのメモリ書き込み傾向を把握するためには、オブジェクト単位での書き込み頻度を収集する必要がある。

ガーベッジコレクション (GC) の戦略として世代別 GC [7] を採用しているオブジェクト指向言語の一部には、その実現のためにライトバリア機構を持つものがある。ライトバリアとは、オブジェクトのデータ書き換えにともなう、ポインタの書き換えを検知する処理である。検知後は、GC アルゴリズムに応じて必要な処理が行われ、その後メモリへの書き込みが行われる。

提案手法では書き込み頻度の収集について、ライトバリア処理を応用し、オブジェクト単位でのメモリ書き込みを検知する。また、得られた特性を基に、NVM, DRAM 間でのオブジェクトの振り分けを行う。

### 3.2 書き込み頻度の記録

収集した書き込み頻度は、言語処理系が持つオブジェクト管理情報に追加して管理する。データの管理方法としては、オブジェクト管理情報と独立したデータ構造を準備し記録時にデータ構造を探索する方法も考えられる。この方法では言語処理系の修正が少なく抑えられる。しかし、ラ

イトバリア処理は高頻度で呼び出されること、オブジェクト数の増加にともない探索のコストが増加していくことなどから提案手法には適さない。

ライトバリア機構を持つ言語処理系では、GCにより不要になったオブジェクトが破棄されてしまう可能性がある。この場合、破棄されたオブジェクトが持っていた書き込み特性は失われてしまう。この場合は、オブジェクトの管理情報と独立して、書き込み頻度を記録しておく必要がある。この際もある程度のコストが必要であるが、GCによる回収はあるオブジェクトに対して1回だけ行われるので、データ構造を探索する必要はない。

### 3.3 書き込み頻度を基にしたオブジェクトの振り分け

提案手法が応用するライトバリア処理は、世代別GCを実現するために言語処理系に実装される機構である。そのため、提案手法は世代別GCをGC戦略として採用している言語処理系を前提とする。世代別GC戦略は、生成されたオブジェクトのほとんどはすぐに参照されなくなるが、ある程度参照が続いたオブジェクトはその後も参照され続ける可能性が高いという経験則 [8] に基づいたGC戦略である。世代別GCでは、オブジェクトは新世代領域と旧世代領域に分けられたヒープの中に配置される。

ほとんどのオブジェクトは新世代領域に生成され、複数回のGCで回収されなかったオブジェクトは旧世代領域に移動される(昇格)。旧世代領域に直接生成される場合もあるが、これは言語処理系の実装に依存している。4章で後述する本研究が実装対象とする言語処理系では、生成対象のオブジェクトがある一定のサイズ以上の場合に、旧世代領域への直接生成が起こる。提案手法では、これらの領域のうち、旧世代領域に配置されたオブジェクトのみを振り分けの対象とし、新世代領域のオブジェクトは対象としない。世代別GCでは、新世代領域をコピーGC方式で管理する。このコピーGCは生存領域を2つの空間に区切り、片方をFrom、もう片方をTo空間とする。新しく生成されたオブジェクトはFrom空間に配置され、GC時には有効なオブジェクトのみをTo空間にコピーする。有効なオブジェクトすべてのコピーが完了すると、FromとToの役割は交代し、GC前にFromであった空間のオブジェクトはすべて無効になる。新世代領域ではこのコピーGCが頻繁に実行され、データのコピーにともなうメモリへの書き込みが頻繁に起こると推定される(図2)。本研究はデータの書き込み頻度に基づいた、DRAM、NVM間でのデータ振り分けを行い、書き込み時におけるNVMのデメリットを隠蔽することを目的としている。そのため書き込みが多く、DRAMに配置されるべき新世代領域はNVMへの配置に適さない。

オブジェクトの振り分けは、旧世代領域が配置されるメモリ空間をDRAMとNVMに分割して割り当て、振り分

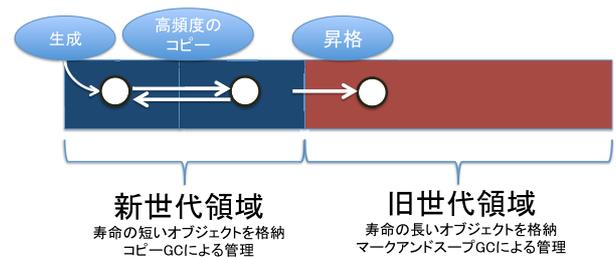


図2 世代別GCの特徴

Fig. 2 Characteristics of generation garbage collection.

け基準を基にオブジェクトを2つの空間の間で、書き込み頻度に基づいて移動させることで実現する。振り分けは旧世代領域でのオブジェクト生成と新世代領域からの昇格時に行われ、その後必要に応じて配置が再決定される。

オブジェクトの生成時、昇格時には、そのオブジェクトの書き込み頻度は明らかでない。そこで、同じクラスのオブジェクトの書き込み頻度に基づいて、配置の振り分けを行う。クラスごとのオブジェクトの書き込み頻度については、一定周期でそれぞれのクラスに対して起こった書き込み回数を計測し、それを基に決定する。

オブジェクト配置時の振り分けのみだと、生成時には書き込みが少なかったが、その後書き込みが増えるオブジェクトに対応できない。また、プログラムの起動直後は、クラスごとの書き込み頻度の情報も十分でないため、プログラム起動直後に生成、昇格されたオブジェクトについては適切な振り分けができない可能性がある。そこで、ライトバリア処理時に書き込み回数がDRAM移動基準を超えていないか調べ、もし超えていたらDRAMへ移動を行う。DRAM移動基準は、旧世代領域での書き込み回数の平均を基に、算出される数値である。この数値は一定周期ごとに更新される。

あるクラスのオブジェクトの中に特異的に書き込み回数が多いオブジェクトがあった場合、そのクラスの書き込み回数が少ないオブジェクトもDRAMに配置されてしまう。前項のようにライトバリア時に書き込み回数を確認することも考えられるが、書き込みが少ないオブジェクトでは、ライトバリア時での検知は難しい。そこで、定期的に旧世代領域のオブジェクトを調べ、NVM移動基準より書き込みが少ないものについてはNVMへ移動させる。NVM移動基準も、DRAM移動基準と同様に、旧世代領域への平均書き込み回数を基に算出する。

## 4. 実装

本研究では、提案手法の評価のために、Ruby [9] の言語処理系である Rubinius [10] を基盤に、Ruby で作成されたプログラムの書き込み頻度を収集する言語処理系を実装した。本章では Rubinius の概要と具体的な実装について述べる。

#### 4.1 Rubinius

Rubinius は、世代別 GC を実現するためにライトバリア機構を実装した Ruby の処理系である。そのため、本研究が提案する手法の実現に適しているため、実装基盤として選定した。本研究では Rubinius 2.0.0 rc1 を基盤に実装を行った。

Rubinius は、実行する Ruby プログラムを仮想マシン (Rubinius VM) が解釈できるバイトコードに変換して実行する。この VM は C++ で実装されている。一方、Ruby プログラムのパースやバイトコードコンパイラは Ruby で実装されており、実行するプログラムと同様に Rubinius VM で実行される。

#### 4.2 書き込み頻度の記録

Rubinius VM 内では、Ruby プログラムが利用するデータがオブジェクトとしてメモリ上に表現される。オブジェクトにはセマンティクスに応じて種類があり、実装上でもそれぞれ異なるクラスとして実装されている。しかし、すべてのオブジェクトはオブジェクト管理クラス (Object-Header) を継承しており、共通のデータ構造を持っている。ObjectHeader に書き込みをカウントするメンバ変数を追加することで、一元的にオブジェクトごとの書き込み回数を記録することが可能になる。また、オブジェクトのセマンティクスごとの書き込み傾向を取得するために、クラスごとの書き込みカウンタもグローバル領域に実装した。

Rubinius では、オブジェクトのメンバ書き換え時にポインタを書き換える。このポインタ書き換え処理を検知し、必要な処理を行うのがライトバリア処理である。本実装ではこのライトバリア処理で、前述したオブジェクト管理クラスの書き込みカウンタを加算する。また、クラスごとの書き込み頻度も記録する。

3.2 節で述べたように、オブジェクトが GC により回収されると、そのオブジェクトに関する書き込み頻度データは失われる。そこで、GC 時に回収されるオブジェクトから書き込みカウンタとアドレス、クラスなどの情報を取得し、独自のデータ構造に記録する。このデータ構造は WriteInformation クラスとして C++ で実装した。WriteInformation クラスはそのインスタンス単体で 1 つのオブジェクトについての情報を保持する。この WriteInformation のインスタンスを管理するリストを、Rubinius VM のグローバルデータ領域に追加した。

収集した書き込み頻度は言語処理系内部で使用するだけでなく、外部に出力し、分析に利用する必要がある。そこで、VM が停止した時点で、生存しているオブジェクトを走査し、それぞれのオブジェクトが保持している書き込み頻度をファイルに出力する機構を実装した。また、Rubinius VM 停止までに GC が回収したオブジェクトについては前述したグローバルデータ領域で管理されているので、この

情報も合わせて出力する。

#### 4.3 GC アルゴリズムの変更

Rubinius は GC 戦略として、複数の GC アルゴリズムを組み合わせる世代別 GC をとっている。Rubinius では、新世代領域の GC アルゴリズムとしてコピー GC を採用している。一方で旧世代領域の GC アルゴリズムとしては、Immix GC とマーク・アンド・スイープ GC の 2 つを採用し、それぞれ別のメモリ空間を管理している。本研究では、実装の単純化のため旧世代領域の GC アルゴリズムはマーク・アンド・スイープ GC のみとし、Immix GC が管理する空間に割当てが起きないように、言語処理系を変更した。

#### 4.4 書き込み振り分け機構

オブジェクトの振り分けについては、3.3 節で述べた手法で行う。DRAM 移動基準と NVM 移動基準は新世代での GC が起こるタイミングで算出し、更新を行う。これは新世代 GC の実行がプログラム全体のメモリアクセス頻度に応じて行われ、周期性を持っていると推定できるためである。また、書き込み回数が少ないオブジェクトの NVM への移動判定も同じタイミングで行うものとする。

なお、本研究では、NVM への書き込みレイテンシや書き込み回数制限は考慮していない。これは本研究の目的が、言語処理系を拡張しプログラムの書き込み頻度を集めることが有効であること、その頻度を基にハイブリッドメモリアーキテクチャでのデータ書き込みの振り分けが有効であることを示すことにあるためである。また、実際の NVM と DRAM 間の振り分けにはメモリ間でのコピーが起こるが、これも再現されていない。あるオブジェクトが DRAM にあるか、NVM にあるかはそれぞれのオブジェクト管理情報にフラグを持たせることで代替した。

### 5. 書き込み頻度の分析

提案した手法が有効であることを示すために、プログラムの書き込み頻度を収集、分析した。本章では、その結果と考察について述べる。収集は実装した言語処理系を Linux で動作させることで行った。以下に環境の概要を示す。

- CPU: Intel Xeon E5345 (2.33 GHz, Quad Core) × 2
- RAM: 6 GB
- OS: Linux 3.4.9 (Gentoo Linux)

今回の計測では、オブジェクトのうち、VM 停止時に旧世代領域に存在しているオブジェクト、VM 停止までに旧世代で回収されたオブジェクトについて分析を行った。3.3 節で述べたとおり、新世代領域は分析の対象としない。ただし、新世代領域で生成され、旧世代領域に昇格したオブジェクトについては分析の対象に含まれる。

ワークロードとして Ruby Benchmark Suite [11] を使用した。Ruby Benchmark Suite は Ruby 向けのベンチマー

クセットである。このベンチマークセットのうち、micro-benchmark として分類されているプログラム群を実行し、書き込み頻度を収集した。

図 3 は収集したオブジェクトへの書き込み回数の分布である。それぞれのオブジェクトを書き込み回数で分類し、それぞれの度数にあてはまるオブジェクトの数を棒グラフでプロットした。横軸が度数で、縦軸が頻度である。縦軸は対数軸である。また線グラフは累積相対度数分布を表している。グラフからは、書き込み回数が 0 から 30 回であったオブジェクトが全体の 98% を超えていることが分かる。このようなオブジェクトは NVM へ置くことが十分に可能であると考えられる。しかしながら一方で、書き込みの回数が多いオブジェクトも少なからず存在しており、1,000 回以上の書き込みを受けたオブジェクトも存在している。このようなオブジェクトは NVM よりも DRAM に置くべきと考えられる。これよりオブジェクトに対する書き込み回数には偏りがあり、書き込み回数による DRAM, NVM 間のオブジェクト振り分けが必要であることが分かる。

図 4 は、クラスごとの平均書き込み回数をプロットしたものである。Rubinius の内部では 68 種類のクラスがあるが、このグラフには平均書き込み回数が 1 回以上であったものについてプロットしている。グラフからは、書き込みが 10 回以下であるクラスがある一方で、一部のクラスのオブジェクトに対しては 1 万回を超える書き込みがあったことも分かる。これよりオブジェクトのセマンティクスに

よって書き込みの回数に偏りがあることが分かる。

以上の分析により、プログラムの実行中に生成されるオブジェクトの書き込み回数には偏りがあること、データのセマンティクスごとに書き込み回数に偏りがあることが分かった。

## 6. 評価

提案手法により収集した書き込み情報が、データの振り分けに有用であることを示すために、4 章で実装した言語処理系のオブジェクト振り分け機構を利用して、旧世代領域に配置されているオブジェクトについて、DRAM, NVM 間でのオブジェクト振り分け実験を行った。ワークロードとしては、5 章と同じ Ruby Benchmark Suite の micro-benchmark を与えて振り分けを実施した。ワークロードはベンチマーク項目ごとにプログラムが分離されているが、この評価ではそれらを順番に続けて実行し、データを収集した。

### 6.1 提案手法の効果

提案手法を用いることで、ハイブリッドメモリアーキテクチャでの NVM への書き込み回数をどの程度抑制できるか予測するために、効果の評価を行った。方法としては、ワークロードを実行し、NVM, DRAM への書き込み回数の変化と、NVM, DRAM 間でのデータ配置サイズの推移を計測した。

書き込み回数の収集にあたっては、旧世代領域に配置されたオブジェクトだけでなく、GC に必要な管理情報も対象にする必要がある。世代別 GC では、新世代領域のオブジェクトへの参照を持つ旧世代領域のオブジェクトを記録している。この記録は記憶集合と呼ばれる。この評価では、オブジェクトごとの書き込み回数に加え、記憶集合への書き込み回数も収集の対象とした。現在の実装では、記憶集合は振り分けの対象とせずすべて DRAM に配置されているため、記憶集合への書き込みはすべて DRAM への書き込みであると見なして評価した。つまり、DRAM への書き込み回数は DRAM に配置されたオブジェクトへの書き込み回数と記憶集合への書き込み回数の和であり、NVM への書き込み回数は NVM に配置されたオブジェクトへの書き込み回数の和である。また、それぞれの領域へのデータ配置サイズを計測する際には、DRAM への配置サイズに記憶集合のサイズを含めた。

オブジェクトへの書き込みはオブジェクトの持つデータの変更にとまなうものだけでなく、新世代領域でのコピー GC や新世代オブジェクトの昇格などの GC 処理にとまなうものもある。これらの参照の書き換えについてもライトバリア処理により検知を行い、参照元の書き込み回数として加算した。また、昇格時にとまなうデータの移動には、NVM または DRAM へのデータの書き込みがとまなうが、

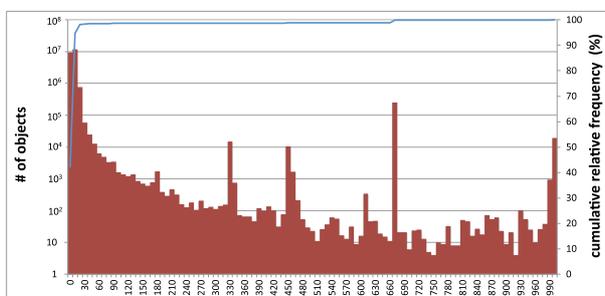


図 3 オブジェクトごとの書き込み傾向  
Fig. 3 A histogram of write access to objects.

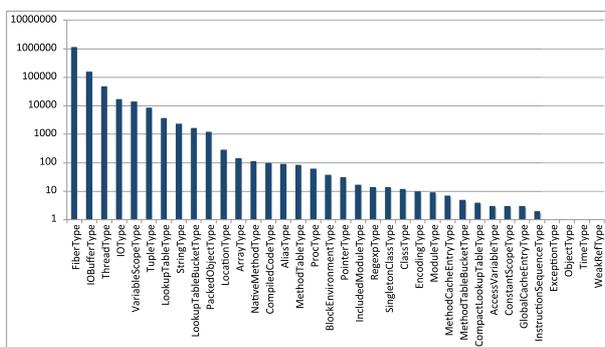


図 4 オブジェクトの種類ごとの平均書き込み回数  
Fig. 4 A histogram of write access to objects (by object class).

表 1 条件設定

Table 1 Experiment parameter.

	条件 A	条件 B
DRAM 移動基準	平均書き込み*1の 50%	平均書き込みの 75%
NVM 移動基準	平均書き込みの 25%	平均書き込みの 75%

表 2 NVM への書き込み数 (単位: 回)

Table 2 Write access to NVM.

	条件 A	条件 B
振り分けなし	1800244777	1581366751
振り分けあり	57874497	94486913

この書き込みに関しても昇格対象のオブジェクトの配置先に応じて、書き込み回数を加算した。

本研究の段階では、オブジェクトを NVM, DRAM 間で移動する場合に、実際に移動対象オブジェクトのコピーは行わない。ただし、他のオブジェクトや記憶集合から移動対象への参照についてはデータの配置アドレスが変更されたものと見なし、その参照の更新を行い、その参照の更新は更新対象へのデータ書き込み回数として記録した。また、移動対象オブジェクトについては、移動後の配置領域に応じて、書き込み回数を加算した。

本実験では、3.3 節で説明した振り分けの基準値により結果が変わってくるのが予想される。そのため、2つの実験条件を設定した。それぞれ条件 A, B とする。それぞれについての振り分けの基準について、表 1 に示す。ここでは旧世代領域に配置されているオブジェクトの平均書き込み回数を旧世代平均書き込み回数と表す。旧世代平均書き込み回数の算出にあたっては、オブジェクトの配置先を区別せず、旧世代領域に置かれたすべてのオブジェクトを対象とする。

両条件について実験の結果、提案手法を用いることで条件 A で 96%, 条件 B で 94% の書き込みが削減できることが明らかになった。表 2 に結果を示す。ここでは、提案手法による振り分けが行われない場合、メモリへの書き込みはすべて NVM に対して行われるものとし、その回数と振り分け実施時の NVM への書き込み回数を比較した。振り分けを行うことで削減された NVM への書き込みは DRAM に対して行われる。

図 5, 図 6 はそれぞれの条件での、NVM, DRAM 間でのデータ配置割合の変化を時系列で示したものである。このデータは一定周期ごとに旧世代領域に配置されているオブジェクトと記憶集合の情報を集め、分析したものである。赤と青で表された面積の割合が、計測時点での NVM, DRAM に配置されたオブジェクトの合計サイズを表している。この割合は左軸に対応している。灰色と白の折れ線は、それぞれ計測周期ごとの DRAM と NVM への書き込

\*1 「平均書き込み」は旧世代平均書き込み回数を表す。

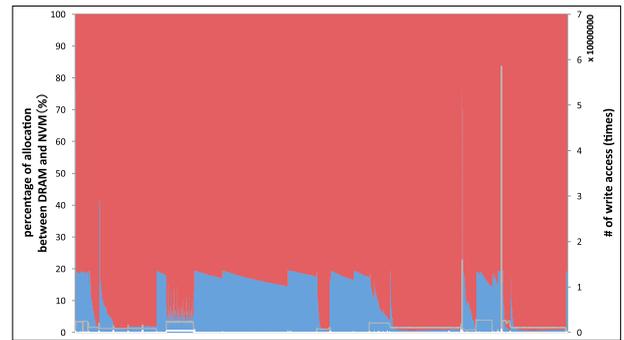


図 5 振り分け結果: 条件 A

Fig. 5 Result of distribution (condition A).



図 6 振り分け結果: 条件 B

Fig. 6 Result of distribution (condition B).

み回数を表している。この書き込み回数は右軸に対応している。図 5 と図 6 から、どちらの条件でも NVM への書き込みは DRAM に比べてつねに低く抑えられていることが分かる。オブジェクト配置については、条件 A では平均して約 89%, 条件 B では 94% を DRAM に配置できた。しかしながら、実行の過程で DRAM への配置割合が条件 A では 90%, 条件 B では 70% を超える場合も確認された。ハイブリッドアーキテクチャを利用する際には、その目的から DRAM の使用量が少ないことが望ましいため、これは問題である。この結果より、提案手法はハイブリッドメモリアーキテクチャでの、NVM 書き込み削減に一定の効果があると評価できるが、DRAM への配分が極端に大きくなる場合があり、このようなケースに対応する必要があることが分かった。

## 6.2 提案手法によるオーバーヘッド

提案手法では、オブジェクトごとの書き込み回数に基づいて旧世代オブジェクトの移動を行う。この旧世代オブジェクトの移動は通常の言語処理系にはないものであり、提案手法を導入する際のオーバーヘッドとなる。本評価では、そのオーバーヘッドとして、オブジェクト移動にともなうメモリアクセスのコストを書き込みと読み込みについて計測を行い、評価を行った。

オブジェクト移動にともなう書き込みコストには、移動

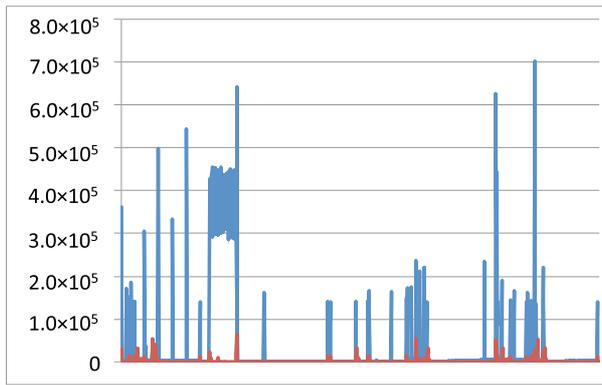


図 7 DRAM に対する書き込みとオブジェクト移動コスト (条件 A)  
**Fig. 7** Write access to DRAM and cost of objects migration (condition A).

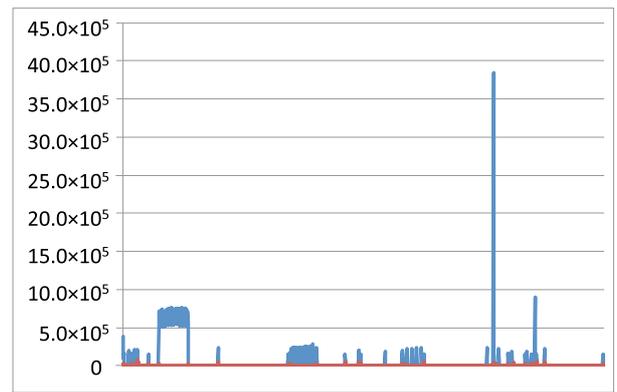


図 9 DRAM に対する書き込みとオブジェクト移動コスト (条件 B)  
**Fig. 9** Write access to DRAM and cost of objects migration (condition B).

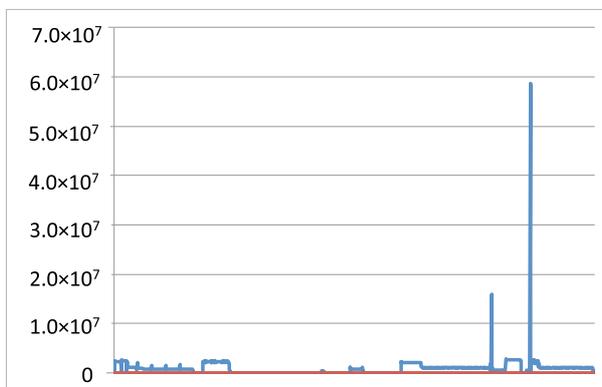


図 8 NVM に対する書き込みとオブジェクト移動コスト (条件 A)  
**Fig. 8** Write access to NVM and cost of objects migration (condition A).

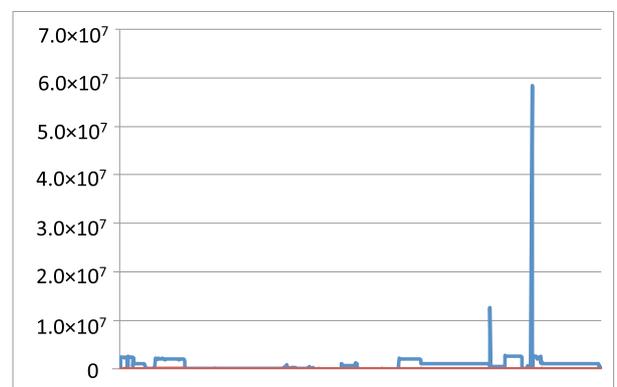


図 10 NVM に対する書き込みとオブジェクト移動コスト (条件 B)  
**Fig. 10** Write access to NVM and cost of objects migration (condition B).

対象オブジェクトのコピーと、移動対象への参照の書き換えがある。前者については、どの領域への移動が起こったか一定周期で計測した。移動対象への参照の書き換えについては、オブジェクトに移動にともない、書き換えが起こる回数を一定周期で計測した（書き換え対象の探索コストについては後述する）。本評価では、この領域ごとの移動回数と参照の書き換え回数の和をオブジェクト移動コストとして定義する。

図 7, 図 8 は、条件 A について DRAM, NVM への書き込み回数とオブジェクト移動コストの変化を示したものである。グラフのうち、青線で示したものが DRAM, NVM それぞれへの書き込み回数であり、赤線で示したものがオブジェクト移動コストである。条件 B については図 9 と図 10 に条件 A 同様に示した。それぞれの図より、ワークロードの実行を通して、オブジェクト移動コストが全体の書き込み回数より低いことが分かる。書き込み回数とオブジェクト移動コストをそれぞれ合算した結果、条件 A のとき、NVM への書き込み約 5,787 万回に対して、オブジェクト移動コストは約 151 万回であった。これは NVM への書き込み回数の約 2.6% に相当する。DRAM への書き込み約

19 億回に対しては、オブジェクト移動コストは 318 万回であった。これは DRAM への書き込み回数の約 0.16% に相当する。条件 B については、NVM への書き込み約 9,449 万回に対して、オブジェクト移動コストは約 180 万回であった。これは NVM への書き込み回数の約 1.9% に相当する。DRAM への書き込み 17 億回に対しては、オブジェクト移動コストは 277 万回であった。これは DRAM への書き込み回数の約 0.16% に相当する。これより、NVM, DRAM それぞれの書き込みに対するオブジェクト移動コストは非常に低いことが分かった。

オブジェクトの移動にともなう読み込みコストとしては、移動対象オブジェクトへの参照を探索するためのメモリアクセスがある。あるオブジェクトが移動するには、そのオブジェクトへの参照を持っているオブジェクトを探索し、参照を移動先に書き換える必要がある。その参照の探索に必要なメモリアクセスが前述したオブジェクトを探索するためのメモリアクセスである。この探索の単純な方法として、存在するオブジェクトをすべて走査する方法があるが、この方法には探索コストが大きいという問題がある。この探索を効率的に行う方法としては、旧世代オブジェクトを

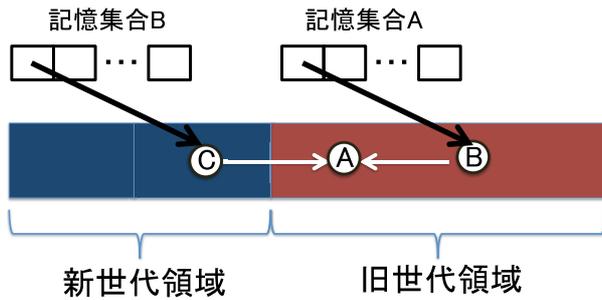


図 11 記憶集合 A, B の概略

Fig. 11 Remembered set A and B.

参照するオブジェクトを記憶する記憶集合を作成し、移動時には記憶集合に登録されたオブジェクトのみを走査する方法がある。しかしながら記憶集合方式をとったとしても、記憶集合への登録数が多ければ効率的な探索を行うことはできない。そこで今回の評価にはどちらの探索方法が効率的か判断することを目的に、実際に旧世代領域への参照を持つオブジェクトを記録する記憶集合を追加し、ワークロードについて記憶集合への登録数を一定周期で計測した。参照元オブジェクトが新世代であるか旧世代であるかによって、参照の傾向が異なる可能性を考慮して、旧世代オブジェクトを記録する記憶集合 A と、新世代オブジェクトを記録する記憶集合 B を追加した。図 11 に計測の概略を示した。図中では旧世代領域に配置されたオブジェクト A に対して、それぞれ参照を持つオブジェクト B, C がそれぞれの記憶集合 A, B に記録されることを示している。計測の結果、記憶集合 A への登録は平均で約 7 万 9 千エントリであることが分かった。一方、記憶集合 B への登録は平均で 43 エントリであることが分かった。特に、記憶集合 A への平均登録数は、同時に記録した旧世代オブジェクトの平均個数と一致しており、ほぼすべての旧世代オブジェクトが記憶集合 A に記録されていることが分かった。この実験より、旧世代領域からの参照を探索する場合は、記憶集合を追加する方式も、旧世代オブジェクトをすべて探索する方式も読み込みコストに差がないことが確認できた。よって、本評価では移動対象オブジェクトへの参照を探索する方法として、旧世代オブジェクトからの参照については旧世代オブジェクトをすべて走査する方法を採用し、新世代オブジェクトからの参照については、記憶集合を追加する方法を採用した。

図 12, 図 13 は、条件 A, B それぞれについて、オブジェクト移動にともなう読み込みコストを一定周期で計測し、示したものである。プロットがない区間は、その区間ではオブジェクトの移動にともなう読み込みコストが発生しなかったことを表している。ここで読み込みコストとは、移動対象オブジェクトへの参照を探索するために、旧世代領域の全オブジェクト、前段で述べた記憶集合 B、通常の記憶集合を走査する回数と定義する。読み込みに関し

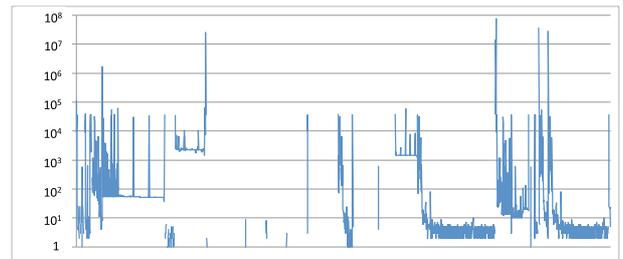


図 12 読み込みコスト (条件 A)

Fig. 12 Cost of read access (condition A).

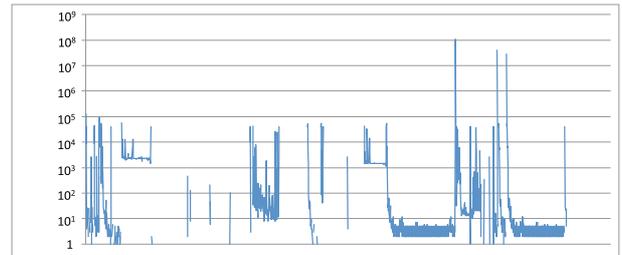


図 13 読み込みコスト (条件 B)

Fig. 13 Cost of read access (condition B).

ては、NVM と DRAM でコストに差はないため、読み込み対象が置かれている領域は区別せずに計測した。図 12, 図 13 からは、条件 A, B それぞれについて最高で約 1 億回の読み込みコストが発生していることが確認できる。読み込みコストを合算したところ、条件 A のときには約 1.9 億回、条件 B のときには約 1.8 億回の読み込みコストが発生していることが確認できた。これより、提案手法のオブジェクト移動には多大な読み込みコストがかかることが分かった。

オブジェクト移動のオーバーヘッドには、メモリへのアクセス回数だけでなく、その実現のために追加された記憶容量もある。今回の手法では、記憶集合 B のデータ量がこれに相当する。そこで、記憶集合 B のデータ量の変化を計測したところ、条件 A, B どちらについても、最大で約 108 KB であることが確認できた。

### 6.3 評価のまとめ

これらの評価の結果、今回対象としたワークロードでは提案手法により、提案手法を利用しない場合に比べて 90% 以上の NVM への書き込み抑制効果が期待できることや、ワークロードの実行を通して、平均して約 89-94% のオブジェクトを NVM に配置できることが確認できた。提案手法によるオーバーヘッドについて評価したところ、オブジェクトの移動についての書き込みコストは、NVM に対する書き込みに対して最大で約 2.6%、DRAM に対する書き込みで最大で約 1.6% 程度であり、書き込み回数全体に対して低い割合であることが確認できた。しかしながら一方で、読み込みコストについては、最大で 1.8 億回発生することが確認できており、オブジェクト移動にともなうメ

メモリ読み込みが提案手法の大きなボトルネックであることが分かった。

## 7. 関連研究

ハイブリッドメモリアーキテクチャにおけるデータ振り分けに関しては文献 [13], [14] の先行研究がある。文献 [13] はハイブリッドアーキテクチャにおいて、ハードウェアから提供される書き込み頻度情報を基に、OS がページ割当ての管理を行う研究である。文献 [14] はページ更新頻度の情報を基に、ページ割当てを DRAM, NVM に振り分ける研究である。いずれもページ単位でのデータの振り分けを議論しており、本研究とは異なる。また、これらは OS レベルでのデータの振り分けを目的としている点からも本研究とは異なる。

データの持つセマンティクスをハイブリッドメモリアーキテクチャの管理に利用する研究として文献 [6] がある。この研究ではページの書き込み間隔に加えて、プロセスのセクション情報やメモリにロードしたファイルの種類などに基づいて、振り分け先を決定する方法を検討している。本研究では、オブジェクト単位でのセマンティクスを利用することを検討しているため、この点が異なる。

## 8. まとめ

不揮発性メモリ (NVM) の実用化研究が進み、主記憶としての利用が積極的に検討されている。NVM を主記憶として取り入れることで、計算機システムの消費電力削減や柔軟なシステム状態の保存と復帰などが可能になる。現状の NVM は書き込みに関していくつかの問題を持っており、それを隠蔽するため DRAM とのハイブリッド構成にすることが検討されている。ハイブリッド構成においては、データの配置先を振り分ける必要があるが、そのためにはそれぞれのデータに対して、どの程度の書き込みが発生するのかの情報 (書き込み頻度) を収集する必要がある。本研究ではプログラムの書き込み情報収集の手段として、言語処理系レベルからのアプローチを提案した。具体的には、一部のオブジェクト指向言語が持つライトバリア機構を応用し、プログラムが利用するデータについて、書き込み頻度を記録、収集する手法を提案した。また、その書き込み特性を基にオブジェクトを DRAM, NVM 間で振り分ける手法についても提案し、評価を行った。結果として、提案手法により、ハイブリッドメモリアーキテクチャにおける NVM への書き込み抑制の可能性を示した。しかしながら一方で、オブジェクトの移動にともなう読み込みオーバーヘッドが大きいなどの性能上の課題も明らかになった。

本研究の段階では、振り分け機構の実装と評価まで実現できており、実際のオブジェクトの移動については実現されていない。そこで次の段階としては、振り分けの結果に応じたオブジェクトの移動を実現し、より精度の高いオー

バヘッドの評価などを行う。提案手法を実用的に評価するには、オペレーティングシステムによる支援が不可欠であるが、そのような支援機構はまだない。そこで、OS と提案手法が用いる言語処理系のインタフェースについても検討する必要がある。以上あげた 2 つが、直近の課題である。

## 参考文献

- [1] Tehrani, S.: Status and Prospect for MRAM Technology, *Proc. 22nd Symposium on High Performance Chips (Hot Chips)* (2010).
- [2] Park, S.W.: Overcoming the Scaling Problem for NAND Flash, *Flash Memory Summit* (2012).
- [3] Lee, B.C., Ipek, E., Mutlu, O. and Burger, D.: Architecting Phase Change Memory as a Scalable Dram Alternative, *Proc. 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pp.2-13 (2009).
- [4] Zhou, P., Zhao, B., Yang, J. and Zhang, Y.: A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology, *Proc. 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pp.14-23 (2009).
- [5] Ramos, L.E., Gorbatov, E. and Bianchini, R.: Page Placement in Hybrid Memory Systems, *Proc. International Conference on Supercomputing (ICS '11)*, pp.85-95, ACM (2011).
- [6] Mogul, J.C., Argollo, E., Shah, M. and Faraboschi, P.: Operating System Support for NVM+DRAM Hybrid Main Memory, *Proc. 12th Conference on Hot Topics in Operating Systems (HotOS '09)*, USENIX Association, Berkeley, CA, USA (2009).
- [7] Jones, R. and Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley (1996).
- [8] Lieberman, H. and Hewitt, C.E.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM*, Vol.26, No.6, pp.419-429 (1983). Also report TM-184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569 (1981).
- [9] Ruby, available from (<http://www.ruby-lang.org/>).
- [10] Rubinius, available from (<http://rubini.us/>).
- [11] Ruby Benchmark Suite, available from (<http://groups.google.com/group/ruby-benchmark-suite/>).
- [12] Qureshi, M.K., Srinivasan, V. and Rivers, J.A.: Scalable High Performance Main Memory System Using Phase-change Memory Technology, *Proc. 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pp.24-33, ACM (2009).
- [13] Dhiman, G., Ayoub, R. and Rosing, T.: PDRAM: A Hybrid PRAM and DRAM Main Memory System, *Proc. 46th Annual Design Automation Conference (DAC '09)*, pp.664-669, ACM (2009).
- [14] Zhang, W. and Li, T.: Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures, *Proc. 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pp.101-112, IEEE Computer Society (2009).
- [15] 中村成洋, 相川 光: ガベージコレクションのアルゴリズムと実装, 第 1 版, 秀和システム (2010).



中川 岳 (学生会員)

2012年筑波大学情報学群情報科学類卒業。現在、筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程にてオペレーティングシステムに関する研究に従事。



追川 修一 (正会員)

平成8年慶應義塾大学より博士(工学)。平成16年筑波大学大学院システム情報工学研究科助教授に着任。現在、筑波大学システム情報系情報工学域准教授。オペレーティングシステムに関する研究に従事。電子情報通信学

会, IEEE 各会員。