東京大学
情報理工学系研究科 創造情報学専攻
修士論文

Flash メモリを併用するスモールデバイス向けメモリ管理システム

A memory management system using Flash memory for small devices

井上 曜
Akira Inoue

指導教員　千葉 滋 教授

2021 年 1 月

概要

本研究では，ガベージコレクションとページングアルゴリズムを改良して，スモールデバイス向けの Flash メモリを併用するメモリ管理システムを提案する．Flash メモリは，安価でプログラムと定数を置くための記憶デバイスで，書き込みの耐性が低く，高レイテンシで一定サイズ単位でメモリにコピーしてから読み書きする必要があるという特徴がある．本システムでは，バーチャルマシンの読み/書きバリアを用いたソフトウェア的なアドレス変換とページングを用いて，Flash メモリを追加の記憶領域とする．そして，Flash メモリへの書き込みを低減するために，ページングのアルゴリズムを改良し，変更のあったページの置き換え優先度を下げる．また，変更のあったオブジェクトが分散しても書き込み回数を低減できるガベージコレクションを提案する．この方式を採用した Scheme 処理系を実装し，提案手法の有用性を実証した．

Abstract

In this paper, we suggest a memory management system that uses Flash memory for small devices by improving garbage collection and paging algorithm. Flash memory is an inexpensive storage device for storing programs and constant objects, has low write tolerance, and has the constraint that it must be copied to memory in fixed size units with high latency before reading and writing. In this system, Flash memory is used as an additional storage area by using software address translation and paging using the read/write barrier of the virtual machine. Then, in order to reduce the writing to the Flash memory, the paging algorithm is improved and the replacement priority of the written page is lowered. We also propose garbage collection that can reduce the number of writes even if the changed objects are dispersed. We implemented a Scheme processing system that adopted this method and demonstrated the usefulness of the proposed method.

目次

第 1 章　はじめに 1
1.1 Flash メモリを併用するスモールデバイス向けメモリ管理システム 1
1.2 本論文の構成 2

第 2 章　記憶デバイスとメモリ管理手法 4
2.1 記憶デバイス 4
2.2 ガベージコレクション (GC) 7
2.3 仮想記憶 10
2.4 ページング 11
2.5 Flash メモリ向けページアウトアルゴリズム 12
2.6 KG-N と KG-W 13
2.7 Crystal Gazer 15
2.8 スモールデバイス向けプログラム言語処理系 17
2.9 Flash を併用したスモールデバイス向けメモリ管理システム 20

第 3 章　Conditional bit LRU と Mark and classified compact 22
3.1 本メモリ管理システムの概要 23
3.2 領域間のオブジェクトの移動 24
3.3 conditional bit LRU 25
3.4 Mark & classified compact 26
3.5 Scheme 処理系 27
3.6 本方式の妥当性 33

第 4 章　実験 37
4.1 実験に使用したアプリケーション 37
4.2 実験の環境 38
4.3 実験結果 38

第 5 章　まとめと今後の課題 40
5.1 まとめ 40

5.2 今後の課題 40

発表文献と研究活動 42

参考文献 43

実験に利用したアプリケーションのソースコード 47

第 1 章

はじめに

1.1 Flash メモリを併用するスモールデバイス向けメモリ管理システム

IoT で使われるような M5Stack や Arduino のようなスモールデバイスの必要性は増加している．このようなスモールデバイスは，汎用的なコンピュータ程の性能は不要だがネットワークとつなげて演算や制御が必要な機械で使われる．近年では，スマート家電やウェアラブルデバイスなどが挙げられる．産業的な用途では，M-duino という ArduinoLeonard ベースの PLC が存在している．このようなスモールデバイスの一つである Arduino UNO は，atmega328p というマイコンを搭載しており，RAM は 2KB で Flash メモリは 32KB，CPU のクロック周波数は 16MHz 程度である．汎用的なコンピュータは，メモリは 1〜128GB，ストレージは数十 GB〜数 TB，クロック周波数は 1〜4GB であり，これと比較してとてもリソースが小さい．Arduino UNO はホビー用途向けであり，同等のデバイスは最低でも 500 円程度で買うことができる．

スモールデバイスは，回路サイズ，価格コスト，消費電力などの制約で使えるリソースが少ないため，アプリケーションの開発は C/C++ 言語のようなオーバーヘッドが少ないものの，高度な機能は使えない言語が用いられている．Micro Python のようなスクリプト言語をスモールデバイスでも動かせる言語処理系は存在するが，スモールデバイスにはメモリに制約があるため，いくつかの高級な機能を使えなかったりスモールデバイスでも搭載しているメモリが大きいものしかつかえないものが多い．そこで，Flash を併用することで仮想的にメモリを多く使えるようなメモリ管理システムを作ることで，スモールデバイスでも高級な機能を用いた開発ができるようにしたい．しかし，スモールデバイスではメモリ管理ユニットが搭載されていないという点や Flash には一定回数書き込むとその領域が使えなくなるという性質によって単純な方法では Flash 領域は仮想記憶として使うことができない．例えば，ATmega328p では書き込み回数は 10,000 回で頻繁に flash 領域にあるオブジェクトを変更させるようなプログラムだとすぐに使えなくなるだろう．このような環境でも Flash を併用してメモリを仮想的に使うことのできるようなメモリ管理システムが必要である．

メモリ管理システムとは，仮想記憶やガベージコレクションを用いて複雑なメモリ操作を抽

象的に自動的に行うシステムである．ユーザにメモリの確保と参照するインタフェースを提供し，複雑なメモリ管理を引き受ける．主記憶装置に加えて補助記憶装置を併用すれば，仮想的に物理的なメモリ空間よりも大きいメモリ空間を使うことができる．仮想的なメモリ空間での操作をユーザが行うのは非常に労力がかかり，メモリ管理システムは MMU 等のハードウェアかソフトウェアでのアドレス変換や主記憶装置と補助記憶装置のメモリのやりとりを効率的に行ない，ユーザは通常のメモリ空間と同じような使い勝手でメモリへの参照や確保を行うことができる．また，C 言語での malloc のような関数を用いて，実行時に新しくメモリ領域の確保が必要になることがある．このように確保したメモリは free 関数等を用いない限り解放されない．メモリ管理システムは不要なオブジェクトを発見し，それを自動的に解放することでユーザは労力や誤った領域の解放や解放漏れによるバグやリソース不足を回避できる．仮想記憶は，汎用的なコンピュータ上では OS やハードウェアの機能によって提供されることが多いが，スモールデバイスでは OS や MMU のようなハードウェアが搭載されていないことが多い．こういったメモリ管理システムを用いてプログラミング言語処理系を実装することで，補助記憶装置を併用するようなプログラミング言語処理系を作ることができる．

本論文では，Flash を併用してもメモリの書き込みをできるだけ抑えることができるメモリ管理システムを提案する．まず，Flash メモリを追加の記憶領域として用いるため，ソフトウェア的なアドレス変換を用いて仮想記憶を使えるようにした．仮想記憶はページング方式を採用し，オブジェクト確保用といった情報や書き込みの記録をもとに Flash メモリへの書き込みが少なくする Conditional bit LRU というページアウトアルゴリズムを実現した．このページアウトアルゴリズムを効率よくおこなうために GC 時の Mark& Compact 時に変更のあったオブジェクトをまとめるようにした．メモリ管理システムを使うには，ランタイムのメモリへの参照バリアと確保関数，動的解放の置き換えが必要であり，それを行えるような仮想機械である Chibi-machine を一から実装した．それから，その仮想機械へのコンパイラも必要であるから，Scheme から仮想機械へのバイトコードに変換するコンパイラである Kallcc も一から実装した．このメモリ管理システムの削減したい値であるページの Flash への書き込み回数と犠牲にしたページフォルト数と同等のページイン回数を実装した Scheme 処理系を用いて測定した．提案手法の書き込み回数は他の手法と比較して最も小さかったが大きな差はでなかった．無駄なオブジェクトの生成がページへの書き込みを増加させており，今後行うコンパイラの最適化でそれらを削除できればある程度差がでると考えている．そして，ページフォルト数はそれの削減に特化した LRU よりも劣るが，Flash の性質である書き込み時間は読み込み時間よりも大きいことから犠牲にしたページフォルト数よりも書き込み回数を減らしたことによる時間のほうが大きいという結果であった．

1.2　本論文の構成

2 章では，本研究の理解に必要な前提知識と本研究で取り組むメモリ管理システムについて述べる．また，不揮発性メモリと DRAM を併用するメモリ管理システムの紹介やスモールデバイス向けの言語処理系についても述べる．

3 章では，提案手法の説明と作成したシステムについての説明を行う．4 章では本手法のメモリ管理システム妥当性を検証するために行った Flash への書き込み回数実験の内容とその実験結果と考察ついて述べる．5 章では本研究のまとめと今後の課題について述べる．

第 2 章

記憶デバイスとメモリ管理手法

本研究は，スモールデバイス用の仮想的に多くのメモリを使うことのできる言語処理系のために Flash メモリと RAM を併用するメモリ管理システムを目指している．Flash メモリと SRAM の組み合わせることを行うため，最初にいくつかの記憶デバイスについて説明する．それから，Flash を併用したスモールデバイス向けのメモリ管理手法の理解のための前提知識を説明する．ガベージコレクション (GC)　，仮想記憶，そして，仮想記憶の手法の一つであるページングについて述べる．次に関連研究として，不揮発性メモリを考慮したメモリ管理システムやスモールデバイス向けの言語処理系を紹介する．最後に提案手法を実現するための問題についても述べる．

M5Stack や Arduino といったスモールデバイスは，回路サイズ，価格コスト，消費電力などの制約で使えるリソースが少ないため，アプリケーションの開発は C 言語のようなオーバーヘッドが少ないものの，高度な機能は使えない言語が用いられている．Micro Python のようなスクリプト言語をスモールデバイスでも動かせる言語処理系は存在するが，スモールデバイスにはメモリに制約があるため，いくつかの高級な機能を使えなかったりスモールデバイスでも搭載しているメモリが大きいものしか使えないものが多い．そこで，Flash メモリを RAM と併用するメモリ管理システムを組み込むことで，安価で低メモリ容量のスモールデバイス上で高級な機能を用いた容易にプログラミングできるような言語処理系を提供したい．しかし，FLash メモリは書き込み方法の制約や，書き込み回数の耐久性が低いため，通常のメモリのように使うことは困難である．

2.1　記憶デバイス

記憶デバイスはデータを保持するためのデバイスである．記憶デバイスは，速度，価格，コスト，面積あたりの容量，消費電力，耐久性等のうちどれかが優れていても，他の要素が他の記憶デバイスより劣る．一般的に，安価な記憶領域は大容量だがアクセス速度が遅く，高価な記憶領域はアクセス速度は速いが容量が小さい．そのため，いくつかの記憶デバイスを階層状に組み合わせて，全体的には理想的な記憶を構築する．上の階層は，高価で高速だが低容量なものを置き，下の階層は安価で低速だが大容量なものを置く．また，上の階層にいくほど

CPU に近く，上の階層は下の階層の記憶領域の部分集合になっている．例えば，上の階層から順にレジスタ，1 次キャッシュ，2 次キャッシュ，DRAM，HDD や SSD のような構成となる．プログラムは，参照の局所性に基づいているため，多くはこの構成でも効率的に動作する．参照の局所性には，時間的局所性や空間的局所性というものがある．時間的局所性はある時点で参照された記憶デバイス上のリソースは，近い将来再び参照される可能性が高いことを意味する．空間的局所性はある記憶デバイス上のリソースが参照されたたとき，その近くのリソースも参照される可能性が高いことを意味する．

記憶デバイスには，電源を供給しないと記憶が保持できない揮発性メモリと，そうでない不揮発性メモリがある．揮発性メモリは，SRAM や DRAM 等の高速な記憶デバイスであるため，主記憶やキャッシュで用いられる．不揮発性メモリは，Flash メモリや HDD のように，RAM ほどの速度はなく，耐久性がないが大容量であるため補助記憶装置に使われる．

2.1.1 DRAM

DRAM は，コンピュータの主記憶に用いられる記憶デバイスである．1 つの記憶セルには，キャパシタ内の電荷がたまっているかどうかで 1 か 0 かを持たせている．電荷へのアクセスにトランジスタを 1 つしか使用しないため，安価で面積あたりの容量も大きい．一定時間がたつとキャパシタから電荷が失われ，記憶が損失するため，内容を読み出して再度書くというリフレッシュという操作が必要である．リフレッシュは電力を消費するため，常にリフレッシュをしている DRAM は消費電力が大きい．

2.1.2 SRAM

SRAM は，キャッシュメモリや Arduino，M5Stack のような低消費電力デバイス向けのメモリ用に用いられる記憶デバイスである．DRAM より高速で情報を読み書きできる．リフレッシュが不要で，わずかな待機電力を流すことで記憶が保持できる．1bit 保持するために，4 から 6 個のトランジスタを使用するため，高価で面積あたりの容量も小さい．

2.1.3 NOR 型 Flash メモリ

NOR 型 Flash メモリは，スモールデバイスのコード領域や定数領域に使われる不揮発性メモリである．バイト単位で読み書きできるが，書き込む前にブロック単位での消去が必要である．また，表 2.1 で示すように，組み込みの NOR FLash であれば，読み込みは DRAM の 10 倍程度だが，書き込みは 1 バイトあたり，100 倍遅い．消去の時間はさらに大きく，1,000,000 倍程度の時間がかかる．書き込みの耐性が低く 10 万回程度書き込まれた領域は使えなくなる．このような書き込み耐性の低さとレイテンシの性質から，単純には RAM の拡張としては使えない．

2.1.4　NAND 型 Flash メモリ

NAND 型 Flash メモリは，SSD や USB メモリ等で使われる不揮発性メモリである．ページ単位での読み込みとブロック単位での書き込みが必要である．また，書き込む前にブロック単位での消去が必要である．読み込みは DRAM の 10 倍程度だが，書き込みは 1,00000 倍遅い．　2.2 のように，NOR Flash より読み込みは遅いが，書き込みは速い．書き込みの耐性が低く 10 万回程度書き込まれた領域は使えなくなる．このような書き込み耐性の低さとレイテンシの性質から，単純には RAM の拡張としては使えない．

2.1.5　相変化メモリ (PCM)

PCM[1] は，不揮発性メモリの一つである．PCM は，バイト単位のアクセス可能で，高密度でスタンバイ電力が低いが，レイテンシが高く，書き込み耐性が低い特徴がある．2.2 のように，アクセスのレイテンシは，DRAM より大きく NAND Flash より小さいため，その間に新たな階層を挟むことことのできるデバイスである．書き込み耐性は Flash メモリよりは高いが，それでも一定回数書き込むと壊れるため，通常のメモリのように使うにはそのまま使うことが出来ない．PCM 用のメモリ管理システムは，レイテンシの差を埋めること [2] や書き込みの耐性の低さを考慮するようなこと [3, 4] が提案されている．書き込み耐性の低さを考慮した PCM 併用のメモリ管理システムは 2.6 節と 2.7 節で紹介する．

表 2.1. 記憶デバイスのスペック [5]

	DRAM	SRAM	Embedded Flash(NOR)	Flash(NAND)	PCM
endurance(cycles)	10^{16}	10^{16}	10^5	10^5	10^9
Read access time(ns)	10	< 1	10	100	10
Write access time(ns)	10	< 1	10^3	10^6	100

表 2.2. Flash メモリのスペック (Cypress)

	NOR Flash(S70GL02GT)	NAND Flash(S34ML04G2)
cost per bit (USD/bit)	6.57×10^{-9}	2.533×10^{-9}
Random Read speed	120ns	30µs
Write speed	*Slower*	*Faster*
Erase speed	520ms	3.5ms

2.2 ガベージコレクション (GC)

プログラムが動的にヒープに割り当てたオブジェクトが不要な場合，その領域を再利用することができる．C 言語の free のような明示的な解放を行うか，ランタイムシステムによって自動的に解放させる方法がある．後者をガベージコレクション (GC) という．

2.2.1 Mark&Sweep

Mark&Sweep は，Mark フェーズと Sweep フェーズからなる GC である．Mark フェーズでは，ルートから辿れるオブジェクト総てにマークをつける．マークを付けられたオブジェクトは生存していると判断できる．ルートとは，コールスタック，レジスタ，グローバル変数領域などのようなランタイムが直接管理している領域である．Sweep フェーズは，ヒープを走査し，マークがついていないオブジェクト，すなわち生存していないオブジェクトを解放するフェーズである．Mark&weep は，Ruby のような汎用的な言語処理系の他，Picobit[6]，Mruby や Micro Python のようなスモールデバイス向けの言語処理系でも用いられている．

マークフェーズでのマークは，確保したオブジェクトのヘッダや，別に確保した専用の領域の 1 ビットを用いることが多い．後者の別に確保した専用の領域にビットを置くことをビットマップマーキングと呼ぶ．前者のようにオブジェクト内のヘッダにマークビットに配置すると，マークフェーズの度にオブジェクトに書き込みが生じてしまうため，オブジェクトへの書き込みが生じると都合が悪いシステムの場合ビットマーキングを用いる．例えば，プロセスの fork のように，データを複製して別のメモリに置くが必要になった時点でコピーせず，変更があった場合のみ複製するようなコピーオンライトという管理方法が存在する．ビットマップマーキングを用いなければ，GC のマークフェーズの度に不要であるはずのコピーが大量に生じてメモリを圧迫するだろう．図 2.1 にビットマップマーキングの例を示す．ヒープには 3 つのオブジェクトがおかれており，各オブジェクトのフィールドとヘッダはすべて 1word であるとする．ビットマップマーキングでは，オブジェクトを配置な位置すべてに対してビットを用意する必要がある．図では，3 つのオブジェクトが置かれているが，ビットマップの必要なビット数はワード分の 10 である．オブジェクトと対応するビットの位置は，ヒープの先頭からのオフセットで得られる．メモリシステムによっては，ヒープをチャンクに分割し，それぞれのチャンクはメモリ上に分散して置かれているようなものがある．そういうシステムでは，チャンクごとにビットマップを作り，それぞれのチャンクの先頭からのオフセットにあるビットを使う．

2.2.2 Mark& Compact

Mark&Compact は，生存しているオブジェクトをまとめることによって断片化が発生しない GC アルゴリズムの 1 つである．Mark フェーズと Compaction フェーズからなる．Mark フェーズは，Mark&Sweep の Mark フェーズと同じである．Compaction フェーズでは，生

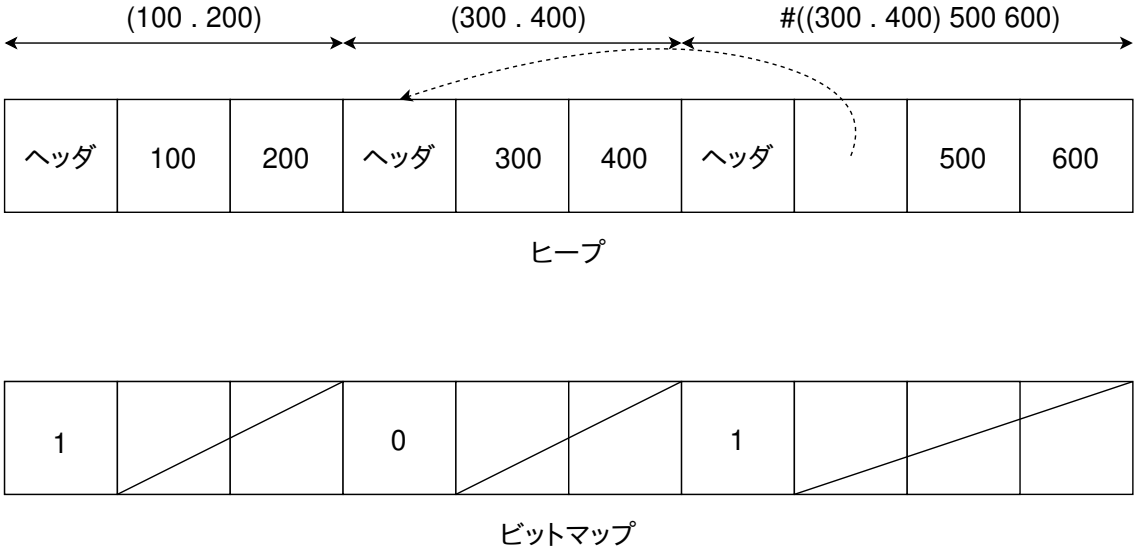

図 2.1. ビットマップマーキング

存しているオブジェクトを隙間がなくなるようにつめて再配置する．このとき，オブジェクトのアドレスが変わってしまう．各オブジェクトのフィールドには古いアドレスを保持しているため，新しい移動先のアドレスに変更する必要がある．

Lisp2 アルゴリズムは，Compact アルゴリズムの 1 つである．移動先のアドレスを保持するために，すべてのオブジェクトにアドレスフィールドとして，追加の 1 ポインタ分の領域を追加する．アルゴリズムは，3 つのフェーズからなる．

1. ヒープ空間を走査し，すべてのオブジェクトの移動先のアドレスを計算し，それぞれの追加のアドレスフィールドにいれる．
2. ヒープ空間を走査し，フィールドにある古いアドレスへの参照をそのアドレスにあるオブジェクトの追加のアドレスフィールドの値に置き換える．
3. すべてのオブジェクトを隙間がなくなるようにつめて再配置する．

Lisp2 アルゴリズムでは，新しいアドレスを保持するために，最低でもポインタサイズ分の追加のフィールドがオブジェクトの個数だけ必要なことが欠点である．利点は，次に説明するような他のコンパクションアルゴリズムと比較して高速であることが挙げられる．

Jonkers のアルゴリズム [7] は，Lisp2 アルゴリズムのような移動先のアドレス保持のための追加領域が不要なアルゴリズムである．　ただし，制約として，オブジェクトヘッダのサイズがポインタのサイズ以上であり，かつポインタが他の値と区別可能である必要がある．このアルゴリズムでは，スレッド化という方法により，あるオブジェクトが自身を参照しているオブジェクトを見つけ出すことのできるような構造に変形する．スレッド化とは，図 2.2 のようにヘッダのアドレスと自身を参照しているオブジェクトのフィールドを線形にポインタでつなぎ，つながれている末尾のフィールドの中身をヘッダにいれる操作である．Jonkers のアルゴリズムでのコンパクションアルゴリズムは次の 2 つのフェーズからなる．

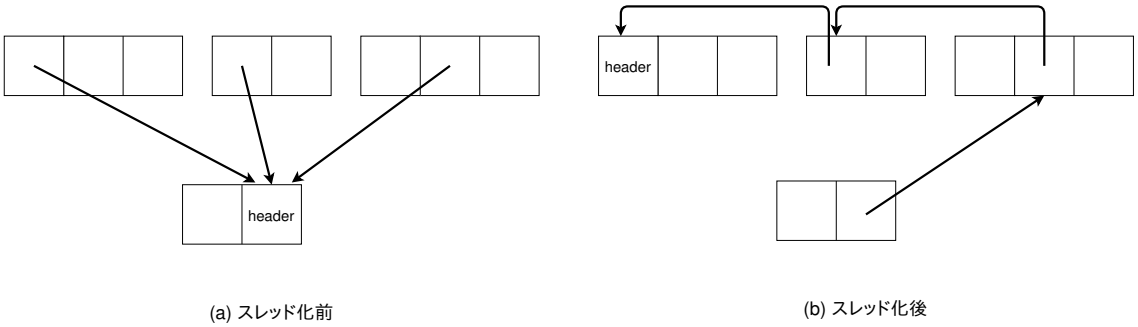

図 2.2. スレッド化

1. ヒープ空間を走査し，マークビットが立っているオブジェクトをスレッド化する．
2. ヒープ空間を走査し，オブジェクトを端につめていく．詰めるときに移動先のアドレスを得られるため，スレッド化で得られた自身を参照しているオブジェクトのフィールドの中身を移動先のアドレスに置き換える．

Jonkers のアルゴリズムは，追加のアドレス保持の領域を減らせるが，スレッド化や置き換えで変更が頻発することとオブジェクトのフィールドを複数回辿るため，キャッシュとの相性が悪いという欠点もある．

2.2.3 コピー GC

コピー GC は生存しているオブジェクトを別の空間につめてコピーすることによってヒープを圧縮しつつ不要なオブジェクトを回収する GC である．このアルゴリズムでは，メモリ空間を 2 つにわけ，一方を from 空間にし，もう一方を to 空間とする．そして，オブジェクトは from 空間にのみ割り当てられる．GC が必要になったとき，ルートからたどりコピーしていく．1 つのオブジェクトを to 空間に移したあとに from フィールドの元にあったアドレスに to 空間への移動先のアドレスを記録しておく．記録されているコピー先のアドレスに置き換え，そうでないものは自身と同様にコピーさせたあとそのコピー先のアドレスに置き換える．to 領域と from 領域という本来必要なメモリ空間の 2 倍の領域が必要な点ではあるが，メモリの断片化が発生せず，Mark&Compact のような GC 時の生存オブジェクトへの参照が多発しないことが利点である．

2.2.4 リファレンスカウント GC

リファレンスカウント GC では，オブジェクトの参照数をそのオブジェクトにもたせ，参照数が 0 になったとき回収する GC である．オブジェクトの参照数はオブジェクトのヘッダに格納しておく．オブジェクトが参照されたときに参照数をインクリメントし，参照が外されたときに参照数をデクリメントする．参照数が 0 になれば不要なオブジェクトであるため，それを回収する．オブジェクトの参照の検出には，すべてのオブジェクトの書き込みを監視するた

めにライトバリアが必要である．

リファレンスカウント GC は，他の GC とは違い GC の計算が一度に集中せずに分散されて行われることが利点である．他の GC では，ヒープサイズが最大に近いと GC が頻発に生じるという欠点があるが，リファレンスカウント GC ではそれはおこらない．一方で，循環参照しているオブジェクトの回収がそのままではできないという問題がある．外部から参照されていないオブジェクト同士が参照しあう，あるいは自身を参照している状態の時でも参照カウントは 1 以上になる．この問題への対処にはルートから辿るような他の GC アルゴリズムを用いてこれを回収する方法がある．汎用的な用途で使われる CPython は GC にリファレンスカウント GC を参照し，循環参照しているオブジェクトは，Mark & Sweep を併用することで回収している．一方で，Python のスモールデバイス向けの実装である Micro Python では GC を Mark & Sweep にしている．

2.2.5　世代別 GC

世代別 GC[8] は，アロケートされてから時間の立っていないオブジェクトとそうでないオブジェクトでグループ分けしてメモリ管理する手法である．世代別 GC 自体は，解放そのものの手法ではないため，グループした領域に Mark&Sweep や Mark&Compact などの解放アルゴリズムを用いる．この手法では，ほとんどのオブジェクトは，生成されてすぐに不要になるということを利用している．生存オブジェクトの探索や，コンパクション等のアドレス張替えを長寿命のオブジェクトに対して繰り返し行うことを避けることができる．通常は，年代のグループをヒープ内で物理的に分割して管理する．マイナー GC という若い世代のメモリ領域を優先的に GC を行ない，十分な期間生き残ったオブジェクトを古いオブジェクトの領域に移動させる．古い世代の領域でも不要なオブジェクトが発生するため，マイナー GC よりも頻度は低いが，定期的にメジャー GC という若い領域を含めた全領域で GC する．

マイナー GC 時に，Mark する GC では古いオブジェクトがみつかるとそのオブジェクトとそのフィールドは辿られない．しかし，古いオブジェクトからのみ辿れる若いオブジェクトが存在することがある．このままでは生存している辿られないオブジェクトも解放してしまうため，そのようなオブジェクトへの参照もコレクタが認識している必要がある．そのため，古いオブジェクトから参照される，若いオブジェクトの位置の集合を記録しておく．これをリメンバードセットという．ライトバリアを用いて，オブジェクトへの書き込み時に古いオブジェクトから若いオブジェクトへの書き込みかを判定してリメンバードセットを構築する．

2.3　仮想記憶

仮想記憶とは，メモリ管理を仮想化することでソフトウェアに都合の良い記憶デバイスのように見えるように提供する技術である．このように仮想化されて提供された記憶装置のアドレスは仮想アドレスという．これに対して，主記憶上の実際に有効なアドレスは実アドレスという．ソフトウェアには仮想アドレスが提供されるため，実際にアクセスするためには実アドレ

スへの変換が必要である．これは，メモリ管理ユニット (MMU) というハードウェアで提供されている．

仮想記憶を用いて，HDD や SSD 等の補助記憶装置を併用することで，主記憶の容量よりも大きなアドレス空間を提供することができる．これには，仮想アドレス空間の一部を補助記憶装置に配置し，必要になれば主記憶装置に読み出すことをしている．仮想アドレス空間の単位をバイト単位で行うと変換のコストがかかりすぎるため，ある程度まとまった単位のブロックでやりとりする．ブロックのサイズが固定のときの仮想記憶の方式をページングと呼び，可変長のときの方式をセグメンテーションと呼ぶ．

2.4 ページング

ページングでは，仮想アドレス空間を固定サイズのページ単位で分割する．あるページが必要になったとき，主記憶にそれがなければページフォルトという割り込みが発生させ，そのページを二次記憶から呼び出す必要がある．これをページインと呼ぶ．また，主記憶にページを置くスペースがなくなったときは，主記憶から補助記憶にページを転送する．これをページアウトと呼ぶ．ページアウトするページの選び方はいくつかアルゴリズムがあり，それらをいくつか説明する．

この節で説明するアルゴリズムでは，ページフォルトが起こる回数を小さくなるように対象のページを選択している．ページフォルトが生じると，ページインのために参照のレイテンシが主記憶装置よりも高い補助記憶装置へアクセスするためページフォルトの回数を少なくすることはパフォーマンスを向上させるためには重要である．

2.4.1 FIFO

FIFO では，置き換えの対象は，主記憶の最も古いものが選択される．管理はキューを用いて行ない，置き換えが必要になると，キューの先頭のページを置き換える．新しくページインするとと，ページをキューの後ろに結合する．このアルゴリズムは実装が単純であるが，仕様頻度を考慮しておらず，パフォーマンスが悪い．また，主記憶に置くページ数を増やすとページフォルト数が増加するという Belady の異常として知られる問題もある．

2.4.2 Second Chance

Second Chance は，FIFO を参照されたオブジェクトがページアウトされにくく改良したアルゴリズムである．ページに参照ビットを 1 つ追加し，ページが参照されたとき，参照ページを立てる．ページアウト時に参照ビットが立っていた場合，そのページの参照ビットをリセットし，最後尾につなぎ，ページアウトできるまでこの操作を繰り返す．

2.4.3　LRU

LRU は，最近使われていないページから優先的に置き換えるアルゴリズムである．単純な実装では，参照順序をリストを用いて管理する．このリストの一方の先端を最も最近参照されたページとし，もう一方の端を最も長い時間参照されていないページとして管理する．リストの最も最近参照されているページ側を MRU と呼び，もう一方を LRU と呼ぶ．新しくページインされるページは，MRU 側の末尾に挿入され，ページが参照される度に MRU 側に移動する．ページアウトされるページは LRU 側から選択する．この実装では，参照の度にリスト上で移動が生じるため，オーバーヘッドが大きいことが欠点である．

LRU の近似的な方法として，参照ビットを用いるものがある．ページごとに参照ビットをもたせ，ページが参照されるとビットを立てる．一定時間おきにすべてのページの参照ビットをリセットする．参照ビットの立っていないページが最近参照されていないページとなる．参照の順序は得られないが，最近参照されていないページは分かるため，それらの内どれかをページアウトすれば良い．

2.5　Flash メモリ向けページアウトアルゴリズム

ページアウトするページがページインされてから主記憶で変更されたとき，その内容を補助記憶に反映する必要がある．変更していなければそのまま捨てることができる．Flash メモリは，書き込み耐性が低く，読み込みより書き込みのレイテンシとエネルギーコストが大きい．LRU のようなアルゴリズムは，変更されたページであるダーティページを書き出すときのコストを考慮していない．そのため，キャッシュミスを減らしつつ，書き込みを抑えるようなページ置き換えアルゴリズムが必要である．

2.5.1　Clean First LRU(CFLRU)

CFLRU[9] は，ダーティページの書き出し時のコストを低減するために，優先的に変更のないクリーンページを追い出す戦略をとる．これだけでは，キャッシュミスの数が大幅に増加するため，リストで実装した LRU を Clean-first region と Working region に分割して，キャッシュミスとダーティページの書き出しを最小限に抑えている．リストの LRU 側を clean-first region とし，MRU 側を clean-first region としている．ページアウトのとき，Clean-first region にクリーンページがあればそれを RAM から排出し，なければ最も使われていないダーティページを書き出す．さらに，Clean-first region のサイズを動的，静的に変更できるようにすることで，キャッスミスと書き込みコストのオーバーヘッドの割合を調整している．clean-first region のサイズを大きすぎるとキャッシュミスが増え，小さすぎると書き込みコストが増える．例えば，最近あまり参照されていないが，書き込みがあるページの割合が多いようなことが生じると，clean-first region を小さくすると LRU リスト上はページがあまり参照されていないが占めるようになり，キャッシュミスが増える．一方で，clean-first

region を大きくすると LRU 上に，書き込みのあるページを優先的に保持できる量が少なくなり，書き込みのあるページが追い出されることで Flash メモリへの書き込み回数が増える．

2.5.2 LRU-WSR

LRU-WSR[10] は，クリーンページとあまり参照されていないダーティページを優先的にページアウトさせるアルゴリズムである．CFLRU には，欠点がいくつかあり，LRU-WSR ではそれを解決している．まず，ウィンドウサイズの設定コストやクリーンページの検索といったオーバーヘッドがある．また，LRU に参照頻度の少ないダーティページが増えるとキャッシュヒット率が低下する欠点もある．LRU-WSR では，参照頻度の少ないダーティページの優先度をクリーンページと同等にし，さらに追加の検索やパラメータの検索を不要にしている．

LRU-WSR では，ダーティページで参照頻度の低いものをコールドダーティページと呼ぶ．コールドダーティページは，ダーティページでかつ，追加に各ページに付与したコールドフラグが立っているかどうかによって判断される．ページインの時，ページはコールドフラグをオフにした状態で MRU の位置に置かれる．コールドフラグがオフのダーティページが LRU の先頭に来た場合，ページアウトの必要時にコールドフラグを立てて再度 MRU の位置に置く．コールドフラグが立っているダーティページとクリーンページはそのままページアウトされる．コールドダーティページが参照され，MRU に移動した場合，コールドフラグをオフにする．

2.5.3 CCF-LRU

CCF-LRU[11] は，ダーティページの優先度の調整だけでなく，クリーンページの参照頻度も順位に反映させたアルゴリズムである．ページは，コールドクリーンページ，ホットクリーンページ，コールドダーティページ，コールドクリーンページの 4 つに分類し，式 2.1 の順序でページアウトされる．

$$\text{コールドクリーン} < \text{コールドダーティ} < \text{ホットクリーン} < \text{ホットダーティ} \tag{2.1}$$

CCF-LRU のシステムは，リストで実装した LRU を LRU-コールドクリーンと混合 LRU の 2 つに分ける．コールドクリーン-LRU には，参照頻度の低いクリーンページを置く．混合-LRU ではダーティページとホットクリーンページを置く．ページの参照が行われると混合 LRU の MRU の位置に cold フラグを立てて移動する．ページアウトが必要になったときは，アルゴリズム 1 のようにページアウトされる．

2.6 KG-N と KG-W

書き込み耐性の低い記憶デバイスである不揮発性メモリの一種である PCM と DRAM を併用した構成のためのメモリ解放手法として，KG-N と KG-W[4] がある．PCM は Flash のように，一定回数書き込むと壊れるため，書き込みを抑えることを目的としたアルゴリズムに

Algorithm 1 CCF-LRU のページ選択アルゴリズム

```
while true do
    if cold-clean-LRU is not NULL then
        ↵first-page(cold-clean-LRU);
    else
        for each page from tail-to-head(mixed-LRU) do
            victim = page
            if cold-flag of victim then
                ↵victim;
            else if clean?( victim ) then
                set-cold-flag(victim);
                reinsert-to-cold-clean-LRU(victim);
            else
                set-cold-flag(victim)
                reinsert-to-mixed-LRU(victim);
```

なっている．PCM と DRAM を併用したメモリ管理では，DRAM によく書き込まれるオブジェクトを配置し，PCM に書き込みが少ないものを配置すれば書き込みのコストが少なくなる．また，オブジェクトが若いものに書き込みの多くが発生するという経験則から，生存期間に応じてオブジェクトの場所を変えるために世代別 GC をベースに配置の最適化をしている．

2.6.1　KG-N

KG-N は，若いオブジェクトで書き込みが多く発生することから，DRAM に若いオブジェクトを割り当てるメモリ管理システムである．図 2.3 のように，DRAM は若いオブジェクト用の nursery 領域用で，PCM には成熟したオブジェクト用の mature 領域，大きいオブジェクト用の large 領域，ランタイムのメタデータ用の metadata 領域という構成になっている．小さいオブジェクトは，最初に DRAM 上の nursery 領域に割り当てられ，一定期間生存すると，GC 時に PCM 上の mature 領域に移動させる．大きいオブジェクトはコピーにコストがかかるため，最初から PCM に割り当て，移動は行わない．ランタイムが使うメタデータ等は PCM 上の metadata 領域に割り当てる．

2.6.2　KG-W

KG-N は，若いオブジェクトへの書き込みは削減できたがのこりの PCM に置かれた成熟したオブジェクトへの書き込みは考慮していない．KG-W では，書き込みの監視を行い，時間が経過したオブジェクトを PCM か DRAM に配置するか選択できる．メモリ構成は図 2.4

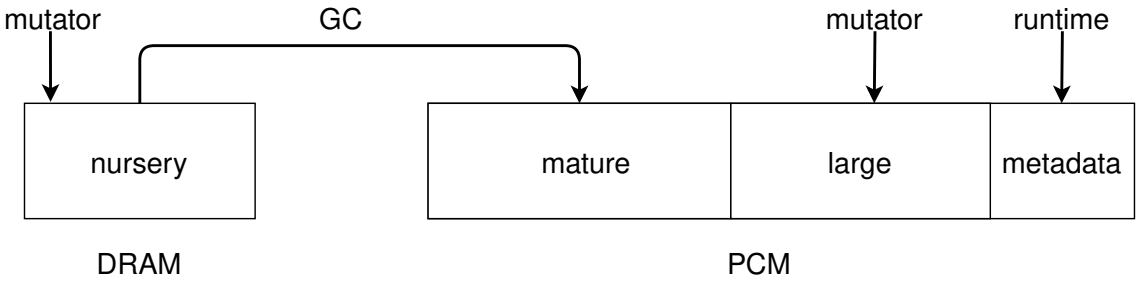

図 2.3. KG-N

のようになっている．若いオブジェクトはまず，DRAM の nursery 領域に割り当てられる．nursery スペースのオブジェクトは，一定期間生存すると，GC 時に DRAM 上の observer 領域に移動する．observer 領域は，nursery の 2 倍の大きさである．observer 領域では，オブジェクトごとの書き込みを計測する．オブジェクトごとに write bit を用意し，書き込みがあればフラグを立てる．observer 領域がいっぱいになると，この領域に対して GC を行う．GC 後に，write bit がオフのものを PCM 上の mature 領域に移動させ，それ以外を DRAM 上の mature 領域に移動する．PCM と DRAM の mature 領域でも observer 領域同様にオブジェクトの書き込みを監視し，全領域に対して GC を行うときに再度最適な場所に移す．PCM 上の mature 領域でオブジェクトの write bit が 1 のとき，write bit を 0 にリセットして，それを DRAM 上の mature 領域に移す．DRAM 上の mature 領域で write-bit が 0 のとき，PCM の mature 領域に移す．nursery 領域と observer 領域の 2 つを用意した理由は，若いオブジェクトは書き込みが多いことだけでなく，すぐ不要になり回収される割合が高いため，nursery で書き込みを監視すると高いオーバーヘッドが発生するからである．また，オブジェクトが DRAM か PCM か選択するまでの時間がかかるようになるため，PCM に置かれるオブジェクトの数が少なくなり書き込みが減少する．

サイズの大きいオブジェクトは最初に PCM の large 領域に割り当てられる．若いオブジェクトは不要になるオブジェクトの割合が高いという性質は，大きいオブジェクトにもおこるため，大きいオブジェクトが DRAM の nurcery 領域空き空間の半分以下である場合は，nurcery 領域に置く．小さいオブジェクトと同様に，nursery 領域で一定期間生存すると DRAM の observer 領域に移動させるが，observer 領域でも生存したオブジェクトは，PCM の large 領域に移動する．PCM の large 領域で一度でも書き込みが行われた大きいオブジェクトは，次のメジャー GC で DRAM の large 領域に移動する．DRAM の large 領域以降は移動は行われない．

2.7 Crystal Gazer

Crystal Gazer[3] は，PCM と DRAM を併用するメモリ管理システムである．PCM は，書き込み耐性が低いため，よく書き込みまれるオブジェクトを DRAM に配置し，書き込まれないオブジェクトを PCM に配置することを行う．このシステムは，実行前にあるオブジェク

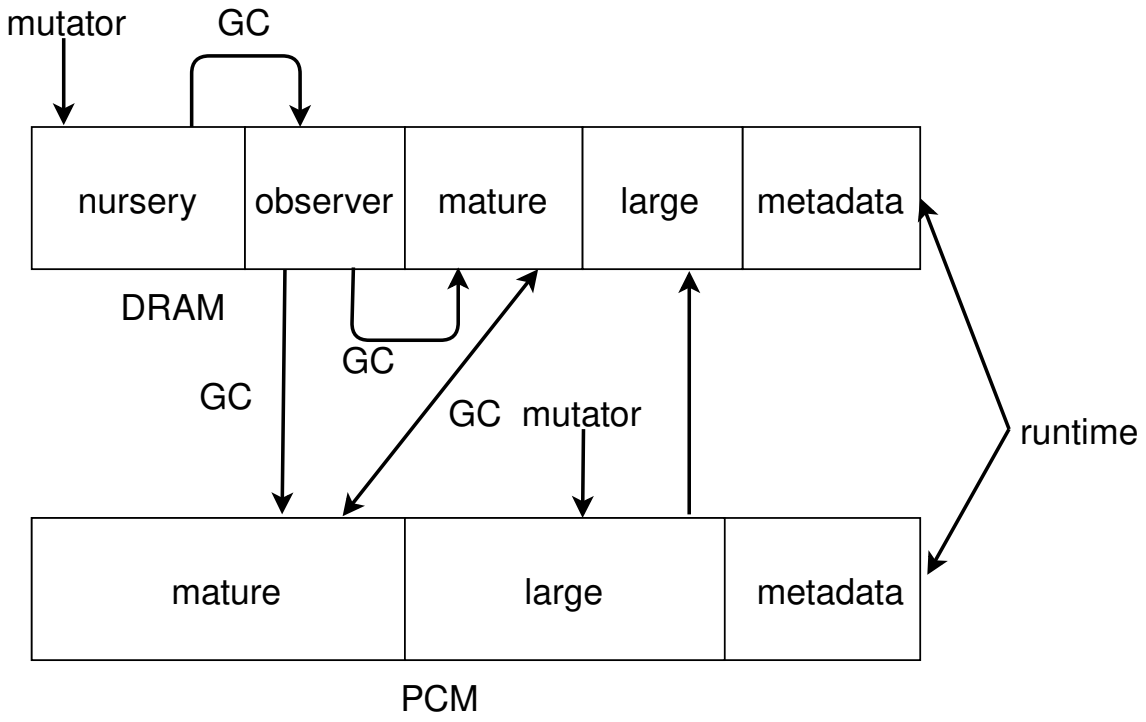

図 2.4. KG-W

トが書き込みが行われるかどうかを予測し，それをもとに PCM か DRAM におくかを判断する．PCM への書き込みを減らすという同様の目的をもつ，KG-N や KGW といった手法では，移動先のような書き込みの検出のための監視や移動先の計算という実行時のコストをかけている．動的な計算はオーバーヘッドがかかるため，パフォーマンスの低下の要因になっている．

Crystal Gazer のシステムは，事前に本番と同じ環境でプログラムを実行し，実行時のプロファイルをとることで配置の予測を行う．実行時のプロファイルでは，各オブジェクトについて次の 4 つの項目を記録する．

1. 区別可能なオブジェクト固有の識別子
2. オブジェクトへの書き込み回数
3. オブジェクトのサイズ
4. オブジェクトを生成したコード上の位置

プロファイルの結果から書き込みの多いオブジェクトを生成しているコードの位置がわかる．あるコードの位置から生成したオブジェクトが書き込みの多い条件を満たしていた場合，そのコードの位置に dram アノテーションを挿入し，書き込みが少なければなにもしない．書き込みの多さの指標は，固定値で書き込み数の判断を行うと PCM への書き込みが減る代わりに DRAM の容量を圧迫するようになるため FREQ と DENS という指標をヒューリスティックなしきい値を選択して使うことで判断している．FREQ は，書き込みの頻度であり，書き込み頻度がしきい値を越える場合ものを書き込みの多いオブジェクトとして判断する．DENS

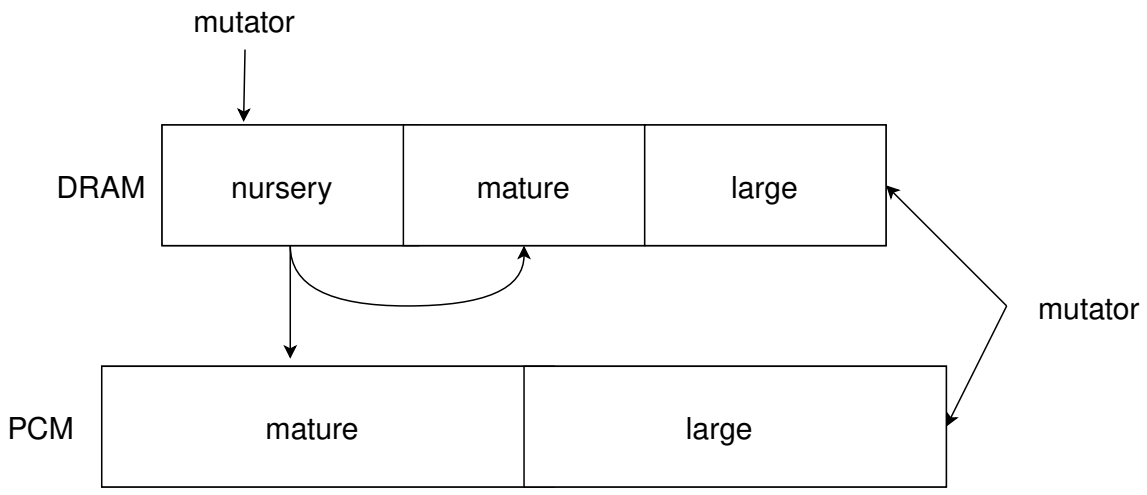

図 2.5. Crystal Gazer

は，書き込みの密度であり，書き込み数とオブジェクトのサイズの比率がしきい値を超えているものを書き込みの多いオブジェクトとして判断する．DENS は，書き込みが集中しやすいサイズの小さいオブジェクトを DRAM に置き，大きいオブジェクトを PCM に置き安くなるため，FREQ よりも PCM の容量を活用できる．

メモリ管理は，世代別 GC を拡張している．図 2.5 に Crystal Gazer のメモリの使いかたを示す．DRAM は，nursery，mature，lage の 3 つに，PCM を mature，large の 2 つに分割する．dram アノテーションの有無に関わらず，オブジェクトを最初 DRAM の nursery 領域に割り当てる．これは，アノテーションに関わらず，生成したオブジェクトの多くはすぐに不要になるためである．そして，nursery で一定期間生き残ったオブジェクトは，dram アノテーションがあるものは DRAM の mature に移動し，それ以外を PCM の mature 領域に移動する．

サイズの大きいオブジェクトは別で管理しており，8kb を超えるオブジェクトを，DRAM アノテーションがあるものは DRAM 上の large 領域に置き，それ以外を PCM 上の large 領域に置く．サイズの大きいオブジェクトも多くはすぐに不要になるため，DRAM の nurcery 領域空き空間の半分以下である大きいオブジェクトはそこに割り当てる．小さいオブジェクトと同様に，nursery 領域で一定期間生存すると DRAM アノテーションがあるものは DRAM 上の large 領域に置き，それ以外を PCM の large 領域に置く．

2.8 スモールデバイス向けプログラム言語処理系

2.8.1 Picobit

Picobit[6] は，スモールデバイスに特化した言語処理系である．システムは，R4RS Scheme を対象としており，PICOBIT 仮想マシン，PICOBIT Scheme コンパイラ，SIXPIC C コンパイラの 3 つの部分からなる．システムの最小セットである PICOBIT Light は，8kB 未満の

RAM と 64kB 未満の ROM で動作し，一番大きい PICOBIT でも 6kB 未満の ROM と 1kB の RAM で動作する．

PICOBIT 仮想マシンは，ガベージコレクション，無制限の高精度整数の実装，およびデータ構造のサポートするバイトコードインタプリタ方式のスタックマシンである．ガベージコレクションは，mark&sweep を用いる．ポインタへの参照を一次的に逆にすることで，マークフェーズにスタックを消費しない Deutsche-Schorr-Waite[12] アルゴリズムを採用して GC に最低限必要なメモリ量を削減している．さらに，PICOBIT 仮想マシンを専用の C コンパイラである SIXPIC C コンパイラを用いることで，汎用の C コンパイラで出力したバイナリよりもコードサイズを削減している．

PICOBIT では，イミュータブルなオブジェクトを ROM 領域に配置することができる．オブジェクトは PICOBIT では 13bit，PICOBIT Light は 8bit の値で表現している．表 2.3 に Picobit のオブジェクトのエンコードの表を示す．値の一部は即値を示し，のこりが ROM と Heap のアドレスと vector 用のアドレスである．ペアやクロージャ等の Heap 上のオブジェクトはすべて 32bit 幅で表現するため，ヒープは 32bit の数値の配列とみなせる．したがって，アドレスはインデックスで表現でき，例えば，260 という値は PICOBIT での最初の ROM の値で，518 は RAM の 6 番目の値を指す．

表 2.3. Picobit のオブジェクトエンコード

Encoding	PICOBIT	PICOBIT Light
0	#f	
1	#t	
2	’()	
3	-1	
4	0	
5-44	1-40	
45-127	41-123	ROM values
127-255	124-251	Heap values
256-259	252-255	N/A
260-511	ROM values	N/A
512-4095	Heap values	N/A
4096-8191	Vector space	N/A

Picobit では，ペアやクロージャ等の Heap 上のオブジェクトは，32bit で表現できる．アドレスや定数を 13bit にしたため，2 つのポインタを持つペアでも 26bit を参照に使い，のこりの 6bit を型情報や GC にマークのヘッダ使えるからである．一般的な言語処理系では，ポインタは 1 ワードとして表現するため，これとオブジェクトヘッダをいれると 96bit 程度のサイズになる．図 2.6 に Picobit のオブジェクトのエンコードを示す．最初のビットが 0 のものは

Integer と Closure があるが，2 番目のビットの 0,1 で区別する．また，最初のビットが 1 のものは，16bit 目から 3bit の値で型を区別する．Integer は，無限長の精度の整数を表現するために使い，16bit 整数の連結リスト状の構造で表現している．to next cell フィールドは次のポインタを指し，のこりの 16bit を整数の値としている．Clsoure は，コードのアドレスと環境の組である．環境は，与えた引数と外側の関数の引数の情報である．String は，to char list に文字のリストをいれることで表現している．to environment は，ペアであるため 13bit に収まり，のこりの 16bit の entry point をアドレスの表現に使っている．U8 Vector は，length を長さのを 13bit の整数で表現し，のこりの 13bit をバイナリをいれる．Continuation は，Scheme の第 1 級オブジェクトの継続である．Gauche のようないくつかの Scheme 処理系では，継続をコールスタックコピーすることで実現している．Picobit は，環境とコードの位置を含んでいる Closure をリストとして繋げることでコールスタックのコピーと同等の方式の継続を実現している．

2.8.2 EveryLite

EveryLite[13] は IOT でのスモールデバイスに特化した言語処理系である．リソースが少ない環境でも動作し，最小限の変更で異なるデバイス上でアプリケーションコードが書くことのできるように作られている．処理系は，軽量のガベージコレクションをもち，組み込み関数を最小限にした小さなフットプリントのインタプリタである．

この言語処理系では，マイクロタスクという単位でプログラムを実行することを前提としたシステムである．マイクロタスクとは，IOT システムでのユーザエクスペリエンスのために重要な，応答待ち時間，メモリとディスクの使用量，エネルギー消費量の 3 つの指標を満たすタスクのことである．IOT システムでのネットワークでつながれたデバイスにクラウドから小さなタスクがデバイスに送信されてくるという状況を想定している．

ランタイムは，インタプリタ以外にもファイルシステムやデバイスドライバ等も含み，それらをモジュールという単位で組み合わせてできている．IOT デバイスにシステムをデプロイするときに，デバイスの仕様や計算タスクに応じて適切なモジュールを選択する．コンパイル時にモジュールを選択する次のようなマクロを提供することでコンパイル時に使うモジュールを決定する．

```
#define AI_MODULE 0
#define WEB_MODULE 0
#define DR_MODULE 0
```

EveryLite のガベージコレクションはプログラムの実行が終わってから動的なメモリスペースのオブジェクトを解放するということを行う．このシステムは，マイクロタスクのみに焦点を当てているため，プログラムの実行中にガベージコレクションが実行されなくても通常どおりタスクを実行できる．インタプリタはプログラムが使用する動的メモリサイズを記録し，サイズがしきい値を超えると，使用されるメモリスペースが大きすぎることを示すエラーコード

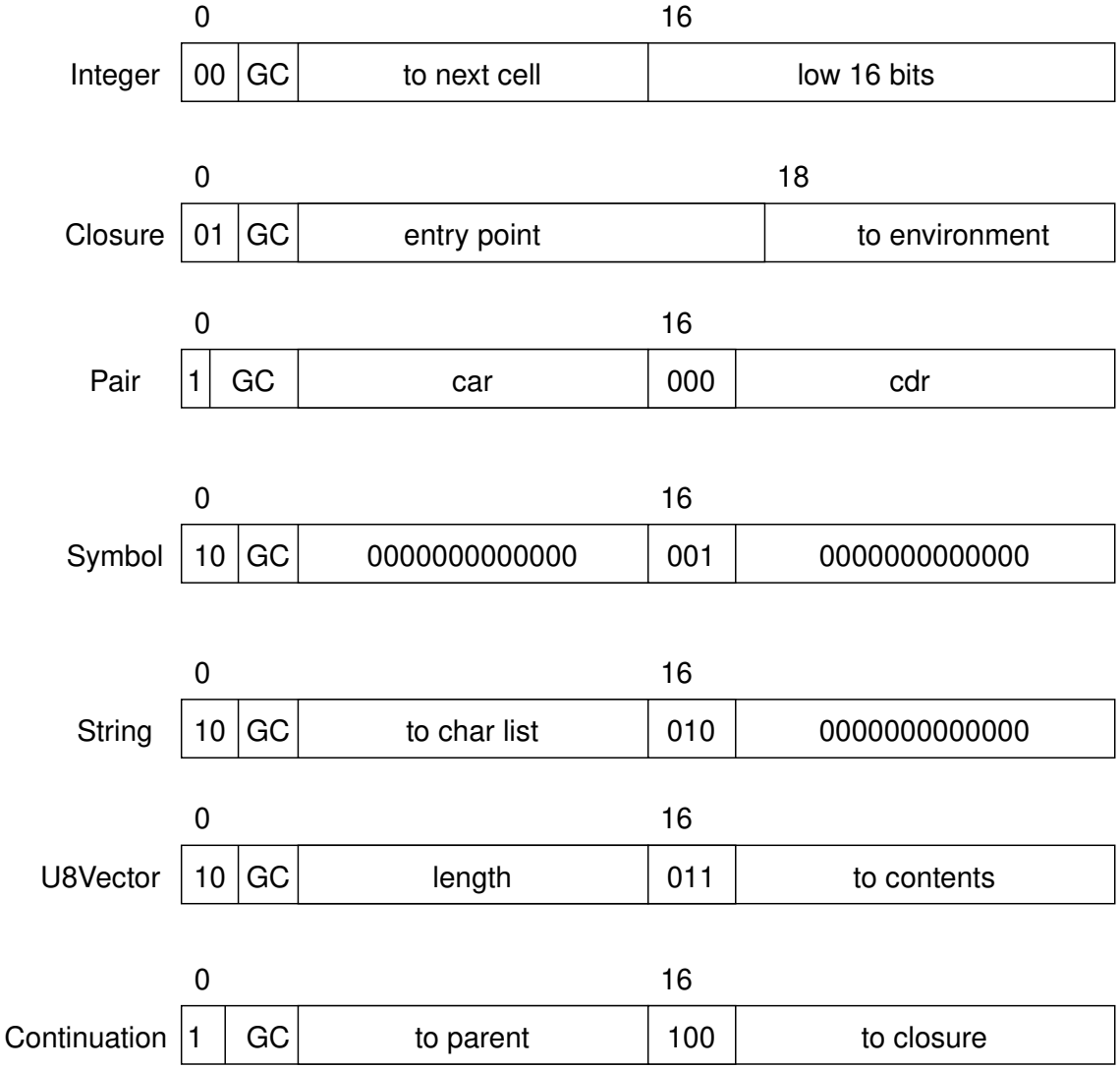

図 2.6. Picobit のオブジェクトエンコード

を返しプログラムを終了する．

2.9　Flash を併用したスモールデバイス向けメモリ管理システム

PCM のような不揮発性の次世代メモリと DRAM 等を組み合わせたメモリ管理システムは存在するが，Flash メモリの性質やスモールデバイスの性質からそのまま導入はできず困難である．まず，Flash メモリは PCM のような次世代メモリよりも書き込み耐性が低く，書き込みのレイテンシが大きい．また，Flash メモリは書き込むときにブロック単位の領域削除が必要であり通常のメモリのように書き込むことができない．さらに，スモールデバイスは RAM がとても小さいため世代数の多い世代別 GC のようなメモリを消費するアルゴリズムは採用できない．

スモールデバイス向けの言語処理系もいくつか存在するが，Flash メモリを併用して仮想的

にメモリ量を増やし，Flash メモリの欠点を軽減するようなメモリ管理システムを採用している処理系は知る限り存在しない．多くは Picobit(Scheme) や JerryScipt(JavaScript)[14], MicroPython(Python)，Mruby(Ruby) といったすでに存在するプログラミング言語用の仮想機械をリソース消費が少ないロジックの採用して実装することや，実装を仕様のサブセットにすることでスモールデバイスでも動作するようにすることしか行われていない．あるいは，IOT 向けといった特定のタスクの性質に特化して，動作するプログラムはマイクロタスクのみとして通常の言語処理系では必要な機能を処理を省いた専用の言語を新しく設計する方針をとっているものもある．

通常のメモリ領域以外の記憶領域である Flash をメモリと同じ用に使うには仮想記憶のような機構を作れば良い．Flash 上のアドレスはメモリアドレスと区別されているため，仮想的なアドレスに置き換えて通常のメモリアドレスのように使える必要がある．一般的な環境で仮想アドレスを実アドレスに変換する場合，メモリ管理ユニットを使うが，IOT デバイスではメモリ管理ユニットを搭載していないものが存在するため，仮想マシン上で read/write barrier を用いて実際のオブジェクトが置かれているアドレスへと変換するといった手法が必要である．このように read/writebarrier が使えるプログラミング言語処理系であればメモリ管理ユニットが存在しないハードウェアでも Flash を追加のメモリ領域でも使うことができる．しかしそれだけでは Flash に一定回数書き込みを行うとその領域が使えなくなるため，現状では書き込み回数が多くなりすぎ実用的ではないことが課題である．

第 3 章

Conditional bit LRU と Mark and classified compact

本論文では，Flash を拡張した言語処理系のためのメモリ管理システムを提案する．page out するページの選択方式として Reference bit LRU を改造したものを用いることと，ガベージコレクション時に compaction することで書き込み回数の低減を図ることを可能にした．

メモリ管理システムとは，仮想記憶やガベージコレクションを用いて複雑なメモリ操作を抽象的に自動的に行うシステムであり，言語処理系の重要な一部である．ユーザにメモリの確保と参照するインタフェースを提供し，複雑なメモリ管理を引き受ける．主記憶装置に加えて補助記憶装置を併用すれば，仮想的に物理的なメモリ空間よりも大きいメモリ空間を使うことができる．仮想的なメモリ空間での操作をユーザが行うのは非常に労力がかかり，メモリ管理システムは MMU 等のハードウェアかソフトウェアでのアドレス変換や主記憶装置と補助記憶装置のメモリのやりとりを効率的に行ない，ユーザは通常のメモリ空間と同じような使い勝手でメモリへの参照や確保を行うことができる．また，C 言語での malloc のような関数を用いて，実行時に新しくメモリ領域の確保が必要になることがある．このように確保したメモリは free 関数等を用いない限り解放されない．メモリ管理システムは不要なオブジェクトを発見し，それを自動的に解放することでユーザは労力や誤った領域の解放や解放漏れによるバグやリソース不足を回避できる．本システムはメモリの確保手続きと参照の手続きを提供し，システムはメモリを不足を検出すると自動的に不要なオブジェクトを解放する．メモリの確保手続に，必要なサイズを渡せば，現在のメモリの使われ方に応じて，Flash メモリか DRAM か最適なものを選択し，実アドレスを仮想的なアドレスに変換したものが得られる．参照手続きに，仮想的なオブジェクトのアドレスを渡せば，RAM 上の実体かコピーされたオブジェクトのアドレスが得られる．そして，本システムの確保と参照手続きを用いてインタプリタやコンパイル方式のプログラミング言語処理系を実装することで，Flash メモリを併用するようなプログラミング言語処理系を作ることができる．

本システムはメモリ上にある容量の小さい volatile 領域と Flash 上にある volatile 領域より大きい non-volatile 領域がある．volatile 領域は容量が小さいがページングを必要とする non-volatile 領域と違い直接参照であるため高速であり変更に制限がないためこの領域に空き

があれば優先的に volatile 領域から確保する戦略をとる．volatile 領域は貴重であることと全領域の頻繁なガベージコレクションは計算コストが大きくと Flash への書き込みが生じるため volatile 領域だけの Mark& sweep のガベージコレクションが必要である．volatile 領域だけのガベージコレクションを使うには，non-volatile 領域に volatile 領域のオブジェクトを書き込みを行った時に仮想マシンの write barrier に remenberd set へ登録する機構の追加によって使用可能になる．non-volatile 領域は仮想マシンの read/write barrier によって必要に応じてページ単位でメモリにページをコピーする (ページイン)．そして，ページをコピーする領域が RAM になくなった場合は優先度の低いページをメモリから取り出し変更が必要であれば Flash へ反映させせる (ページアウト).

提案手法としてページアウト時に書き込み頻度を下げるためにメモリ管理を工夫した．まず，書き込みのあるページをできるだけメモリ上に保つためのページアウト対象の選択アルゴリズムを改良した．さらに変更のあるオブジェクトだけをメモリ上にあるコピーされたページ内に保つためにガベージコレクション時の compaction で変更のあったオブジェクトをまとめるような機構も作成した.

3.1 本メモリ管理システムの概要

本メモリ管理システムでは，ヒープを 2 つの領域に分ける．1 つが RAM 上の volatile 領域で，もうひとつが flash 上の non-volatile 領域である．RAM には，volatile 領域以外にも後述するページを置くための領域と GC 用のメタデータ用のスペースも置く．図 3.1 に各領域の使われ方を示す.

オブジェクトの確保には，本システムが提供する動的確保関数の必要なサイズを引数として，与えることで行う．動的確保関数を呼ぶと，指定したサイズ分の領域を volatile 領域に割り当てられる．volatile 領域に空きがなくなると，volatile 領域だけの GC(volatile-GC) を行う．volatile-GC 時に書き込みの少ないものを non-volatile 領域に移動する．volatile 領域から確保できなくなると，non-volatile 領域からも割り当てる．non-volatile 領域のオブジェクトは，仮想的なアドレスが割り当てられており，volatile 領域のアドレスとは区別可能である必要がある．例えば，すべてのオブジェクトの末尾 2 ビットをタグとして使うことにすれば，末尾ビットの 1 つを volatile と non-volatile の区別用に用いればよい．non-volatile 領域のアドレスは仮想的なものなので，有効なアドレスに変換するために，メモリを参照時にバリアを用いてアドレス変換関数を呼ぶ必要がある．non-volatile 領域はページ単位で分割されており，non-volatile 領域上のオブジェクトが必要になると，ページ単位で RAM の専用の領域にコピー (ページイン) し，アドレス変換関数によってコピー先のページ上のアドレスを返す．ページを置くための RAM 上の領域は有限であるため，新しくページを置く前にシステムは自動的に最適なページを選択して RAM から追い出す (ページアウト)．そして，RAM においてから変更があれば，その内容を Flash に反映する.

volatile 領域だけの部分的な GC(volatile-GC) をサポートするために，non-volatile 領域から volatile 領域への参照を作った場合はそのオブジェクトのアドレスの remenberd set を

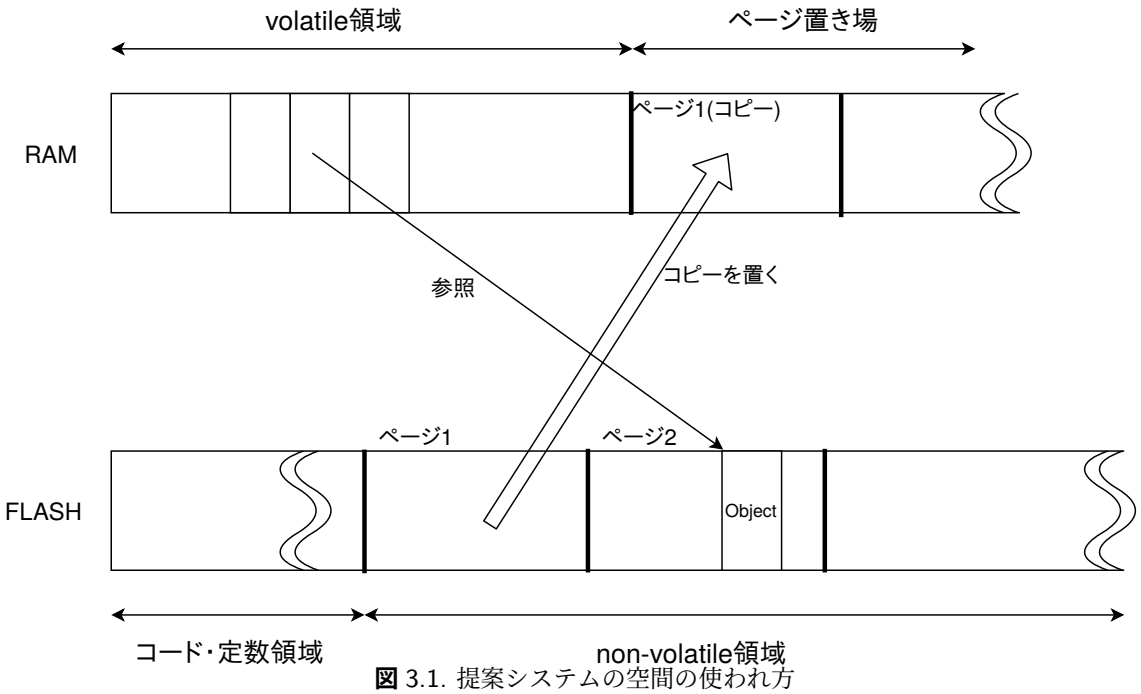

図 3.1. 提案システムの空間の使われ方

作っておく．non-volatile 領域の空きがなくなった時にシステムは全体の GC を行う．このとき，non-volatile 領域のオブジェクトは書き込みの多いものと少ないもので分けてそれぞれ別のページにいれる．また，volatile 領域に半分以上の空きがあった場合，書き込みの多いオブジェクトを non-volatile 領域に移動する

3.2　領域間のオブジェクトの移動

non-volatile 領域にすぐに解放させるようなオブジェクトを置くと，全体のガベージコレクションまで残留し，無駄な Flash への書き込みが生じる．また，non-volatile 領域上に書き込みの多いオブジェクトが増加すると，ページ置き場にダーティページがたまりやすくなり Flash からの読み込みが多発する．そのため，定期的に条件を満たすオブジェクトを volatile 領域と volatile 領域で移動することで，この問題を回避する．

どの領域に配置するのが最適であるかは，生存期間と書き込みの有無と移動時のポインタの張替えのコストをもとに判断する．オブジェクトのヘッダのうち，2bit を書き込みビットとし，1bit を生存ビットに使う．書き込みビットは，オブジェクトの確保時に 0 で初期化し，オブジェクトに書き込みがあれば 1 にする．書き込みビットは，オブジェクトの確保時に 0b00 で初期化され，書き込みが行われる度に 0b01，0b10,0b11 と変化する．0b11 になった後はこれ以上変更はなく，non-volatile 領域に移動してもリセットしない．cons や vector 手続きのようなオブジェクトを確保して，値を書き込むような操作でのオブジェクトへの書き込みには書き込みビットを変更しない．書き込みビットは，今後変更される可能性の判定に使うため，ほとんどのオブジェクトに生じる最初にフィールドに初期値を書き込みの記録はあまり意味

がないからである．生存ビットは，オブジェクトの確保時に 0 で初期化され，volatile-GC を行うときに 1 に変更する．生存期間の情報の記録は，volatile 領域のみで行う．non-volatile 領域に移動した後は volatile 領域に戻らない限りは不要であること，bit がオブジェクト内のヘッダにあるため，ただ長く生存するだけで bit が変更され，Flash への無駄な書き込みを避けるためである．オブジェクトヘッダにはまだ未使用なビットがあるため，生存ビットを複数にすることでより詳細な生存期間情報を得たりすることは可能だが未実装である．

volatile-GC 時では，生存した volatile 領域上の書き込みビットが 0 で生存ビットが 1 のオブジェクトを non-volatile 領域への移動対象としている．ただし，そのオブジェクトが non-volatile 領域上のオブジェクトから参照されている場合，そのオブジェクトが移動すると Flash への書き込みが生じるため対象から外す．Flash から参照が行われているかどうかは，volatile-GC 用に作られたリメンバードセットを用いて判定する．そして，volatile-GC 時の終了時に移動対象のオブジェクトをページ置き場上のオブジェクト確保用のページに移動する．その後，volatile 領域をスキャンし，volatile 上のオブジェクトのフィールドに移動元のポインタがあれば移動先のポインタに置き換える．

全領域の GC 時では，non-volatile 領域と volatile 領域の双方にオブジェクトが移動する．volatile 領域上の書き込みビットが 0 で生存ビットが 1 のオブジェクトを non-volatile 領域への移動する．また，non-volatile 領域上の書き込みビットが 0b11 でかつ volatile 領域の空きの半分以下であれば volatile 領域に移動する．

3.3 conditional bit LRU

ページアウトでは，基本的にはページミスを低減するようにページを選択しなければならないが，書き込みのあったページのページアウトは Flash の寿命を縮めるため，それも避ける必要がある．書き込みが起こるページアウトは，set-car!や set-vector!などによって変更されたオブジェクトを含んだページと新規割り当て用のページアウト時に Flash に反映させる必要のあるページの 2 つで生じる．この 2 種類のページの追い出しの優先度を落としつつ，ページミスを低減するようなページアウトアルゴリズムが必要である．reference bit LRU を改良した，conditional bit LRU を提案する．reference bit LRU は，LRU の近似的な手法である．各ページに 1 ビットを用意し，参照が行われると立てる．一定時間おきにこのビットをオフにする．ページアウトが必要なときにこのビットを見ることで最近参照されていなか判断できる．

conditional bit LRU では，ページに対して，読み書きしたときに立つ参照フラグと書き込みを行った write フラグと新規当て用のページ用のフラグの 3 つを用意する．参照フラグは，参照される度にオンになり，ページの内で参照フラグがオフのもので write フラグがオフで新規割り当て用でないものをページアウトする．この時点でページアウトされなければ，write フラグがオフで新規割り当てページでもないものをページアウトする．ここで，ページアウトされたかに関わらず，すべてのページの参照フラグをオフにする．この時点でページアウトされなければ，write フラグと新規割り当てページと割り当てられていない区間の割合を元に

図 3.2. conditional bit LRU ページアウトの順序

最近参照されていない書き込みのないページ → 書き込みのない新規割り当てでないページ →
書き込みのない新規割り当てページ → 書き込みのあった新規割り当てでないページ →
新規割り当てページで空き領域が多いページ

ページアウトの対象を選択する．

cons や vector 手続きのようなオブジェクトを確保して，値を書き込むような操作でのオブジェクトへの書き込みには write flag を変更しない．具体的には図 3.2 のような順序でページアウトされる．

ページの半分より大きいサイズのオブジェクトを連続で確保すると，ページ置き場が，余裕のある新規割り当て用のページで満たされてしまう．この場合に，ページインしたときにページを置く空間が少なくなり，ページイン，ページアウトが多発し，性能が低下する．そのため，ページ置き場に 3 つ以上の新規割り当てページが存在する場合は，各フラグに関わらず，一番古い新規割り当て用ページをページアウトする．

3.4　Mark & classified compact

3.2 のページアウトの最適化は変更のあったページと変更のないページが別れていなければどれをページアウトしても Flash への書き込みが生じるため意味がなくなる．したがって，今後変更が生じるオブジェクトとそうでないものを分けてページに配置する必要がある．これは，過去に変更のあったものは今後も変更されやすいという仮定から，Mark & compact やコピー GC のようなメモリの再配置を行うガベージコレクションと連動して再配置時に過去の変更記録をもとに今後変更されやすいオブジェクトをまとめることで解決できると考えられる． コピー GC はメモリを使える量が半分になり，スモールデバイスで仮想的にメモリを増やすという目的に合わないため，Mark & compact が良い．そこで mark & compact の改良として，non-volatile 領域のための Mark & classified compact を提案する．

mark フェーズでは，Flash への書き込みをしないために，ビットマップマーキングを用いて，mark ビットを RAM 上に配置することでページは読み込みだけで済む．compaction フェーズのアルゴリズムをアルゴリズム 2 に示す．non-volatile 領域の各ページの生存しているオブジェクトを過去に書き込みが発生したかに基づいて 2 種類に分ける．そして，書き込みが発生していないオブジェクトから flash の各ページに書き出し，その後書き込みが発生したものを flash の各ページに書き出す．実装では，すべてのオブジェクトを読むまで書き出しを待つ必要はない．1 ページ分の容量分のオブジェクトをどちらかの Queue に存在する場合は書き出すようなことをすることで，書き出し待ちの 2 つの Queue によるヒープの使用量を低減することができる．

Algorithm 2 non-volatile 領域用の compaction algorithm

```
writtenQueue ← 空の Queue
noWrittenQueue ← 空の Queue
for page = flash の各ページ do
    for obj = page の各オブジェクト do
        if alive?(obj) and changed?(obj) then
            Push(writtenQueue, obj)
        else if alive?(obj) and not changed?(obj) then
            Push(noWrittenQueue, objectPtr)
addrTable ← 空の辞書
for page = flash の各ページ do
    if not empty?(writtenQueue) then
        while do
            obj ← pop(writtenQueue) または obj ← pop(noWrittenQueue)
            if obj が page にかけない then break
            write(page, obj)
            set(addrTable, obj の旧アドレス , obj の新アドレス ))
for obj = すべての object do
    for member = 0tosizeof(object) do
        if haveKey(addrTable, obj[member]) then
            obj[member] ← addrTable[obj[member]]
```

この compaction アルゴリズムでの新しいアドレスの保持する方法は，Lisp2 アルゴリズムのオブジェクト内の追加ヘッダで持つ手法を RAM 上にに古いオブジェクトアドレスと新しいアドレスのペアを持つように置き換えたものである．non-volatile 領域上のオブジェクトにヘッダを置くと新しいアドレスをセットする時に Flash への書き込みが発生してしまうことと，追加の領域をもつことで volatile 領域だけで動かすときにメモリが不足するためである．Jonkers のアルゴリズムのように，追加の領域を必要としないコンパクション手法も存在するが，キャッシュとの相性が悪いことと複数回オブジェクトへのポインタ張替えが生じるため，Flash のような書き込み耐性の低いメモリデバイスでページングを行うメモリ管理システムで用いるには問題がある．

3.5 Scheme 処理系

評価のため，提案したメモリ管理システムを使った scheme 処理系は Scheme 用のバーチャルマシンである Chibi-machine とそのコンパイラである Kallcc からなる．将来的には開発時

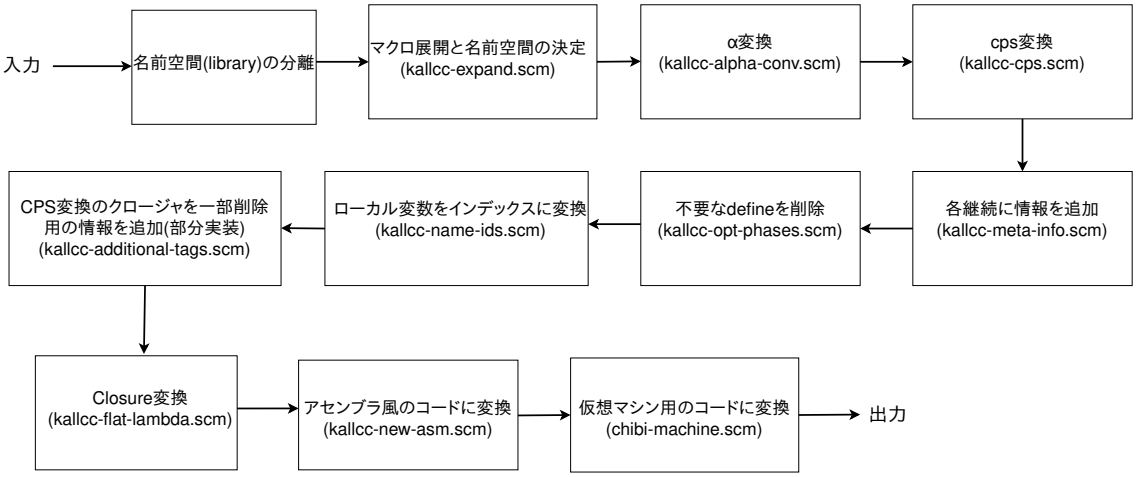

図 3.3. Kallcc コンパイラの各パス

点での最新仕様である Revised 7 Report on the Algorithmic Language Scheme (R7RS) に準拠させるつもりだが，現在は R7RS subset 程度の機能である．R7RS は，約 20 の syntax と約 300 の手続きを提供している．本システムはコアな部分の syntax と手続きは実装した．未搭載の syntax はソースコード 3.1 のようにマクロを用い，手続きもソースコード 3.2 のようにコアな関数を組み合わせれば使うことができる．

Listing 3.1. scheme のマクロの例

```
1 (define-syntax let ;simple-version
2   (syntax-rules ()
3      ((_ ((name val) ...) body1 body2 ...)
4      ((lambda (name ...) body1 body2 ...) val ...))))
```

Listing 3.2. scheme の関数定義の例

```
1 (define (caar pair) (car (car pair)))
```

3.5.1　Scheme **コンパイラ** Kallcc

Kallcc は，本研究のために一から実装した Scheme コンパイラである．Kallcc は Scheme のソースコードを入力し，中間コード，アセンブラを経てターゲットのバーチャルマシンで実行可能なバイトコードに変換してそれを出力する．この変換は多段階のフェーズに分かれている Multi-pass compiler となっている．各フェーズでは，切り出し可能な最小単位となるように実装されており，最適化のフェーズを追加等のあとから必要な変換を容易に挿入しやすいような設計となっている．図 3.3 に Kallcc コンパイラのコンパイルパスを示す．不要な define の削除やある程度の不要な Closure の生成を削除するフェーズがあるが，まだ多くの最適化の余地が残っており，さらにいくつかのフェーズが追加が必要である．

コンパイルする過程の中間コードは継続渡しスタイル (CPS) を用い，そのまま逆変換することなくアセンブラに変換する．継続渡しスタイルは，ソースコード 3.3 のようにプログラムのすべての継続を手続きとして陽に埋め込むような形式である．本実装では，すべての手続きの第一引数に継続を表現した lambda 式を挿入している．次の継続が存在しない場合は，第一引数に'() を挿入する．

Listing 3.3. CPS 変換の例

```
1 (cons (cons 1 2) (cons 2 3))
2 => (cons (lambda (k1) (cons (lambda (k2) (cons '() k1 k2)) 2 3)) 1 2)
```

Scheme では継続は第一級オブジェクトであり，この継続渡しスタイルによって手続きとして表現された継続を用いて実装した．継続渡しスタイルで継続を実装する方針は，Chicken Scheme や Orbit[15] 等のいくつかの処理系で行われている．また，Scheme には末尾呼び出しの最適化も仕様で実装しなくてはならないが，CPS 変換されたコードは手続きが返る機構は不要であり，末尾呼び出しは goto に変換できるため，末尾呼び出しの仕様も満たしている．CPS 変換したままマシンコードに変換すると，ソースコード 3.3 のように，closure の生成が多くなるという欠点はあるが，本システムは仮想的にメモリを増やすことを行っているため，ある程度この欠点は低減されている．ソースコード 3.3 のような手続きの対象が明らかな場合は，静的最適化で中間の継続生成は省くことが可能だが，それは未実装である．

アセンブラはソースコード 3.4 のような表現になっている．これは，(cons 1 2) を CPS 変換した (cons '() 1 2) をアセンブラとして出力した例である．対象のマシンは無限大のレジスタをもつレジスタマシンである．個々の命令は，リストで表現されており，最初の要素は命令でのこりは引数である．引数の (R N) と言った式は，N 番目のレジスタを意味し，(C N) は数値の N を意味する．ペアやベクタ，クロージャ等はレコードというポインタの列として表現し，その型をタイプ用のフィールドにセットしている．レコードの変更命令は，初期化かどうかによってオブジェクトへの書き込みビットの変更の挙動が変わるため，RECORD-SET! と RECORD-SET-INIT!と区別している．呼び出し規約は，0 番レジスタに手続きをセットし，2 番レジスタ以降に引数を順番にいれる．この例では，手続きは継続なしの'() がセット 0 番レジスタにセットされている．クロージャの環境の展開コードは，呼び出し先の手続きの本体の直前に挿入されている．

Listing 3.4. アセンブラの例

```
1 (SET-NULL! (R 0))
2 (SET! (R 1) 1)
3 (SET! (R 2) 2)
4 (RECORD (R 3) (C 3)
5 (RECORD-TYPE-SET! (R 3) (C 1))
6 (RECORD-SET-INIT! (R 3) (C 0) (R 1))
7 (RECORD-SET-INIT! (R 3) (C 1) (R 2))
8 (SET! (R 2) (R 3))
9 (CALL (C 1))
```

最後にこのアセンブラをバイト列に変換し，仮想機械に渡して実行する．

その他の命令は，表 3.1 に示す．命令-引数という命名規則となっている．アセンブラの引数の型ごとに，対応する命令のバイトコード表記を用意している．型の名前は，SMALL-INTEGER は 1 バイトの数，REGISTER はレジスタ番号，2-INT は 2 バイトの数，4-INT は 4 バイトの数となっている．多くの命令は最低限必要なものだが，car や cdr のようによく使われる手続きに対応する命令は，レコードの第一メンバや第二メンバを取り出す簡略化した命令も含まれている．

3.5.2 仮想マシン Chibi-machine

Chibi-machine は，本研究のために一から実装した，提案手法であるメモリ管理システムを搭載した仮想マシンである．Chibi-machine は，無限大の個数のレジスタを持つレジスタマシン方式の仮想マシンとして実装した．1 つのレジスタのサイズは実行する環境の 1word 分としている．仮想マシンが扱える値は，レコードへの参照，26bit の数値，文字，Boolean，空リスト である．レコード以外はそれをエンコードした 1word 長のビット列を使い、参照を扱うときは、タグがついている仮想アドレスとして表現する。各値の末尾 2 ビットを情報ビットとしてもち，1 つを volatile と non-volatile の区別に使い，もう 1 つをレコード用かそれ以外に使った．レコード以外のオブジェクトのエンコードは，末尾 2 ビットの次の 4 ビットに型の種類をつめ，のこりをデータの内容として使う．

図 3.5 にレコード以外のエンコード方法を示す．整数は，型のビットが 0000 で，残りの 26bit を整数の値として使う．加減算はそのまま計算し，積や商，剰余では，整数を 6bit 右シフトしてから計算する．文字型は，型のビットが 0010 で，残りの bit のうち 8bit を文字の値に使う．真偽値や空リストは，型のビットが 0001 で，残りの bit に固有の値を入れる．真は 0b00 で，偽は 0b01 で，空リストが 0b10 である．

レコードは，ペアやベクタ，バイトベクタ，クロージャといったオブジェクトを表現するようなポインタの列である．図 3.4 にレコードのエンコード方法を示す．レコードの最初の要素には，型情報や書き込みフラグ，マークビットを含んだヘッダを配置し，残りはオブジェクトのフィールド用である．ペアは，2 つめのフィールドに car の要素を，3 つめのフィールドに cdr の要素を置く．ベクタは 2 つめのフィールドにベクタ長を，残りのフィールドにベクタの要素を置く．クロージャは，2 つめの要素に環境をおき，3 つめの要素に飛び先のアドレスを置く．クロージャの環境は，外側の関数 1 つに対し，与えた引数をベクタに入れたものをリストでつないだものである．

3.5.3 参照バリアとアドレス変換の実装

仮想マシンで扱うアドレスは，仮想的なタグ情報を含んだアドレスであるため，参照バリアを用いてすべての参照時に仮想アドレスを有効なアドレスへ変換する必要がある．本研究で

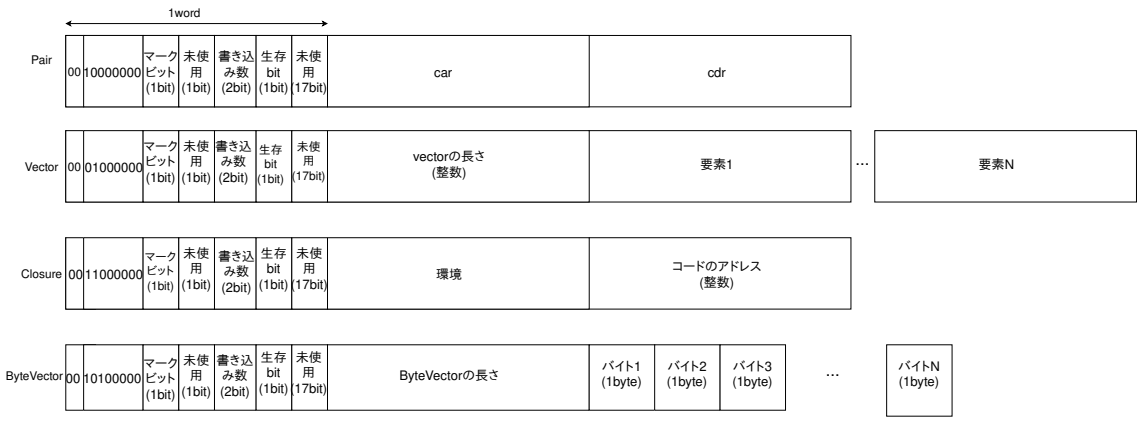

図 3.4. レコードのエンコード方法

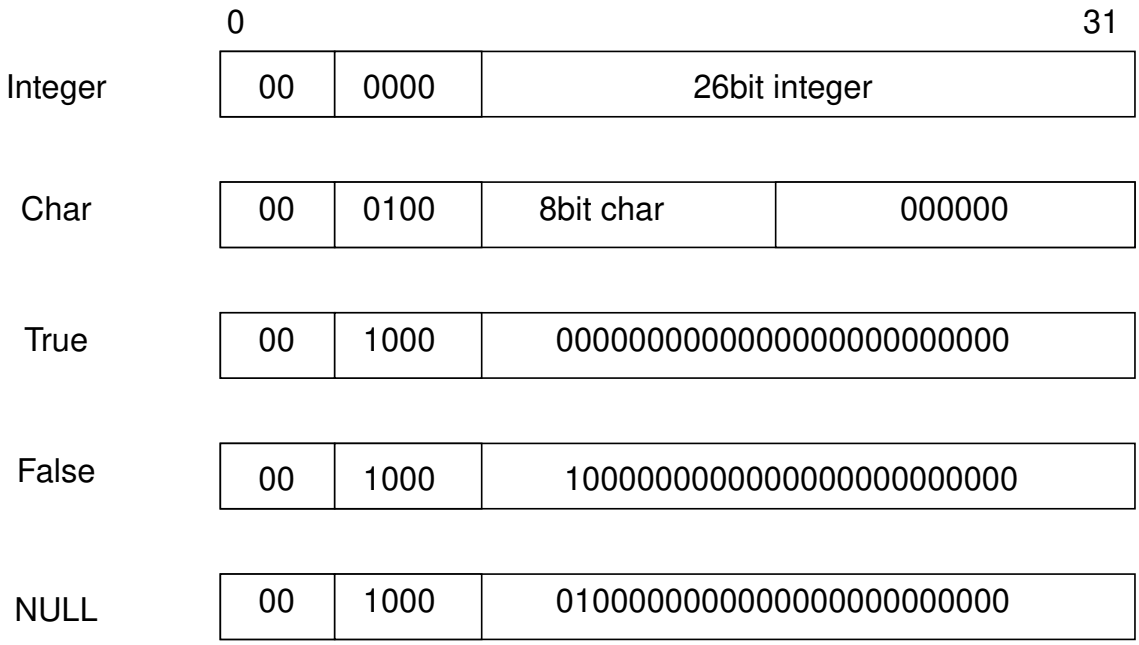

図 3.5. レコード以外のエンコード方法

は，ソースコード 3.5 で示すような ref_addr 関数をすべてのオブジェクトに対する読み込みと書き込み時に呼び出すような実装している．ref_addr 関数の引数の virtual_addr は参照したい仮想的なアドレスで，flash_pages は SRAM に載っているページインされた non-volatile 領域のページ情報の配列で，write_flag は書き込み時のアドレス変換を行うかどうかのフラグ情報である．まず，仮想的なアドレスの末尾ビットから，non-volatile 領域か volatile 領域かを判定し，volatile 領域であれば仮想アドレスと実際の SRAM のアドレスの末尾 2bit 目が 1 になったものなので末尾 2bit をオフにして返す．仮想的なアドレスが non-volatile 領域であれば，対象のアドレスが載ったページが SRAM に載っているか調べ，なければページインする．そして，SRAM に載っているページのコピーの SRAM 上のアドレスがページ情報の cache メンバにあるので，Flash 上のアドレスとページの先頭のアドレスの差分のオフセットを cache

アドレスに加えて SRAM にコピーしたオブジェクトのアドレスを返す．non-volatile 領域上を指す仮想アドレスは，Flash のアドレスの末尾 1 ビットが 1 のものなので，REMOVE_TAG マクロのよに末尾ビットを削除すれば Flash 上のアドレスを得ることができる．最後に，書き込みの記録は，classified compact やオブジェクトの移動ポリシーに必要なので，write_flag が指定されていればオブジェクトとページに書き込みのフラグを立てておく．

Listing 3.5. 参照関数

```
1  #define FLASH_ADDRESS_P(virtual_addr) ((virtual_addr)&0b01)
2  #define REMOVE_TAG(virtual_addr) ((virtual_addr) & (~0b11))
3
4  uintptr_t ref_addr(uintptr_t virtual_addr,Flash_page *flash_pages,char
       write_flag){
5      if (FLASH_ADDRESS_P(virtual_addr)){
6          Flash_page hit_page = search_page_on_ram(virtual_addr, flash_pages)
               ;
7          if (!hit_page){
8              hit_page = page_in(REMOVE_TAG(virtual_addr));
9          }
10         uintptr_t res = hit_page->cache
11         + (REMOVE_TAG(virtual_addr) - hit_page->block_start);
12         if (write_flag){
13             hit_page->write_flag = 1;
14             add_write_flag(res);
15         }
16         return res;
17     }else{
18         return REMOVE_TAG(virtual_addr);
19     }
20 }
```

3.5.4 Scheme 処理系の最適化の余地

本研究で実装した Scheme 処理系である，Kallcc，Chibi-machine は，まだ速度の改善の余地や無駄なクロージャを作らないようにするような処理が残っている．コンパイラは CPS 変換を用いてそれをバイトコードに変換する方式をとっているため，最適化しないと大量にクロージャが生成される．改善の余地は他の Scheme 処理系である Gauche-0.99 と比較して，本処理系が追加のメモリを得るためにバリアやアドレス変換を行っているとはいえ，10 倍程度の差がある．表 3.2 は，Gauche-0.99 を Intel Core i5-4570，メモリ 32GB のマシンで 37 番目のフィボナッチ数の計算で比較したときの時間である．使用したコードは，ソースコード 3.6 に示す．漸化式 $a_n = a_{n-1} + a_{n-2}, a_1 = a2 = 1$ を再帰的に計算する実装となっている．現状の改善の必要のある箇所は，無駄なクロージャの生成や，無駄なレジスタ転送の削減，ガ

ベージコレクションのオーバーヘッドである．

Listing 3.6. ベンチマークのフィボナッチ数列の計算コード

```
1 (define (fib x)
2   (if (< x 2)
3      x
4      (+ (fib (- x 1)) (fib (- x 2)))))
5
6 (fib 37)
```

3.5.5 対応デバイス

実機での動作は，Aruduino Uno 程度のスペック (表 3.3) で動作することが目的ではあるが，現状は開発のしやすさから M5Stack Basic (表 3.4) で動作するものになっている．多くのライブラリは共通のものを用いることができるため，Arduino での実行は Flash 周りのコードを書き換えれば動かすことができる．

3.6 本方式の妥当性

3.6.1 スモールデバイス向けアプリケーション

本提案手法のメモリ管理システムを搭載した言語処理系アプリケーションは，条件がそろうと重いタスクを実行するが，それ以外のときはほとんど計算リソースを消費しないようなものを対象としている．例えば，平常時はセンサーで外部の環境を監視することだけを行ない，なにか変化が生じた時にアクションをおこしたり，記録や送信を行うために複雑な計算をするようなアプリケーションである．平常時は volatile 領域のみで小さな計算を実行し，ガベージコレクションも volatile 領域だけのものだけで行う．そして，メモリを多く消費するような複雑な計算を行うときに non-volatile 領域を併用する．重いタスクを実行するときは，本システムを採用しなくても CPU を使うため時間がかかる．そのため，non-volatile 領域を使うことによる Flash メモリのレイテンシや全領域のガベージコレクションが生じることによるオーバーヘッドによる計算時間が生じてもユーザエクスペリエンスは大きく低下しない．本手法を用いることで，複雑な計算に必要なメモリ量にあわせて容量の大きい RAM を使う必要がなく，平常時に必要な分だけの RAM を乗せるだけですむ．これによりデバイスの価格コストを下げることができ，RAM が DRAM であれば待機時にエネルギーを使わない分エネルギー消費量も抑えることができる．

3.6.2 本方式の汎用性

本方式はバーチャルマシンや事前にネイティブコンパイルする言語において read/write-barrier と動的確保部分を置き換えることで実現できる．バーチャルマシンでの実装は本研究

で実現できた．ポインタを直接操作しないような言語用のコンパイラで提案手法を実現するにはまず提案手法のメモリ管理を組み込む必要がある．そして，メモリを動的確保する関数を提案手法のものに置き換えることと，オブジェクトを参照する時に read/write barrier を挿入するようなコードを出力して non-volatile 領域かを判定すればよい．

表 3.1. 命令セットの一覧

EXIT	仮想マシンを終了する
JMP-{REGISTER, SMALL-INTEGER, 2-INT}	指定したコード位置にジャンプ
JA-REGISTER- {SMALL-INTEGER, 2-INT}	指定したレジスタの中身が真なら指定したコード位置にジャンプ
SET-BP- {SMALL-INTEGER, 2-INT}	指定したアドレスを BP レジスタに入れる
CALL	第一レジスタが closure であれば, closure 中のアドレスにジャンプし, 第一レジスタが null(継続なし) であれば, BP レジスタのアドレスにジャンプする
MV-REGISTER-REGISTER	レジスタの中身を別のレジスタにいれる
SET-NULL-REGISTER	レジスタに null を入れる
FXNUM- {SMALL-INT , 4-INT, 2-INT}- REGISTER	レジスタに fixnum を入れる
{ADD, MUL, SUB, MODULO, QUOTIENT}-{ REGISTER-REGISTER , SMALLINT-REGISTER}	指定したレジスタの値と指定した値で演算を行ない 結果を元のレジスタの値に上書きする
{LESS, GREATER, EQ}-{ REGISTER-REGISTER , SMALLINT-REGISTER}	指定したレジスタの値と指定した値で比較し, 結果をブーリアン型で元のレジスタに入れる
ALLOCATE-RECORD-{ SMALL-INT-REGISTER , REGISTER-REGISTER}	指定したサイズのレコードを動的確保し 指定したレジスタに入れる ここで適宜 GC も行う
NOT-REGISTER-REGISTER	レジスタの値が偽なら真で上書きし, それ以外なら偽で上書きする
RECORD-REF-{ REGISTER-SMALL-INT-REGISTER , REGISTER-REGISTER-REGISTER}	レジスタにレコードのメンバを入れる
RECORD-REF-TYPE-REGISTER	レジスタにレコードのオブジェクトヘッダを入れる
RECORD-REF-FIRST- REGISTER-REGISTER RECORD-REF-SECOND- REGISTER-REGISTER	レコードの 1 番めか 2 番めのメンバをレジスタに入れる
RECORD-SET-TYPE- REGISTER-REGISTER	オブジェクトヘッダをレコードにセットする
RECORD-SET-TYPE-CORE- REGISTER-SMALL-INT	型番号からオブジェクトヘッダを生成してレコードにセットする
RECORD-SET-{ SMALL-INT-SMALL-INT-REGISTER, REGISTER-REGISTER-REGISTER }	レコードのメンバを更新する
RECORD-SET-INIT-{ REGISTER-SMALL-INT-REGISTER, SMALL-INT-SMALL-INT-REGISTER, REGISTER-REGISTER-REGISTER}	初期化のためにレコードのメンバを更新する (書き込みのフラグをページやオブジェクトに記録しない)
RECORD-SET-FIRST- REGISTER-REGISTER RECORD-SET-SECOND- REGISTER-REGISTER	レコードの 1 番めか 2 番めのメンバを更新する

表 3.2. Gauche0.99 との比較

Gauche	2.55 [s]
Kallcc,Chibi-machine	27.57 [s]

表 3.3. Arduino UNO のスペック

CPU	16MHZ
RAM	2KB
FLASH	32KB
EEPROM	1K

表 3.4. M5Stack Basic のスペック

CPU	240MHZ
RAM	520KB
FLASH	16MB

第 4 章

実験

本研究では，conditional bit LRU と Mark and classified compact が妥当であることを検証するために，実装したメモリ管理システムをもつ Scheme 処理系で小さなソフトウェアを実行し，Flash メモリへの書き込み回数と SRAM へのページイン回数を計測する実験を行った．提案手法とそれを別のアルゴリズムに置き換えたものを書き込み数を比較した．実験した結果では，提案手法で Flash メモリへの書き込み回数が減少してはいるが大きく差はでないという結果だった．

4.1 実験に使用したアプリケーション

実験に使うアプリケーションとして，k-means アルゴリズムを用いて画像を減色させるプログラムを使用した．想定として，スモールデバイスで条件を満たしたので重いタスクを実行したときに動くようなプログラムを想定している．画像は，すべての画素が 1 つのページに乗りないことから，ひとつの塊にならないように，縦 × 横 ×3 色の 3 重の vector として表現している．アルゴリズムは，指定した数の代表色を計算し，各画素がいくつかの代表色に近いものを選択するものである．　いくつかの代表色は，代表色 ×3 色の二重の vector として表現している．大まかなアルゴリズムは次の通りである．詳細なアプリケーションプログラムは付録のソースコードに示す．

1. 代表色の初期値を，ランダムで求める．
2. 各画素に対して，3 色との差をそれぞれとり，その 2 乗の和が最も小さい代表色に割り当てる．
3. 各代表色に割り当てた画素の色の平均を計算し，その平均の色に置き換える
4. 2-3 を一定回数繰り返す
5. 画像の各色を割り当てた代表色に置き換える

本実験では，繰り返し数を 10 で，画像のサイズは 16×16 としている．画像のどの画素も選ばなかった代表色は，平均を更新せずにそのままにする．オブジェクトの書き込みを行っている部分は代表色の置き換え部分と各色の平均を求めるために代表色に割り当てられた画素の各色

の総和を更新する部分，そして最後の画像の更新である．平均を求めるための色の和の計算結果は，代表色の置き換え後に不要になり回収され，画像の各色の置き換えはプログラムの最後に行われる処理であるため，提案手法の測定に影響が大きい部分は，代表色の更新部分である．

4.2　実験の環境

本システムは，スモールデバイスである M5Stack と pc 上でも動作する．実験は，Flash メモリと同等のインタフェースをもったヒープ上の仮想的な non-volatile 領域を用いて，pc 上で行った．スペックは，Intel Core i5-4570 の CPU を搭載し，メモリが 32GB で，OS は 64bit の Ubuntu 20.04 である．Flash メモリへの書き込み数を計測するために，計測用の変数を使って，仮想的な non-volatile 領域へ書き込まれる度に計測用の変数をインクリメントする．volatile 領域のサイズと non-volatile 領域のページのサイズとページ数，そしてページ置き場のサイズを表 4.1 のように設定した．仮想マシンのアドレス長は，実行環境の 1word と同じになる．実験の環境では，8byte である．

表 4.1. 実験の設定

volatile 領域のサイズ	1024byte
ページのサイズ	512byte
ページ数	512
ページ置き場のサイズ	4

比較する手法は，提案手法とコンパクション方法とページングアルゴリズムを変えた次の 4 つである．

- 提案手法: Mark and classified compact と conditional bit LRU
- Mark and compact: Mark and compact と conditional bit LRU
- all CFLRU : Mark and classified compact と すべての領域を clean-first にした clean-first LRU
- LRU: Mark and classified compact とページングアルゴリズムを LRU にしたもの

4.3　実験結果

4.3.1　書き込み回数

実験結果を表 4.2 に示す．この表は，それぞれの手法での Flash への書き込み回数である．提案手法が最も Flash への書き込み回数が少ない．Mark and compact にしたもの，ページングを CFLRU にしたもの，LRU の順に書き込み回数が多くなる．書き込み回数が非常に多いのは，クロージャや一時的なデータのアロケート数が多すぎて volatile に収まらずに

表 4.2. 実験結果

提案手法	26813
Mark and compact	27182
all CFLRU	27372
LRU	27553

non-volatile で確保と回収が頻発しているからである．また，差が生まれなかった要因として，今回実行したプログラムでは，書き込みを行うオブジェクトの間にあるオブジェクトはクロージャや一時的なデータほとんどなので，通常の Mark and ompact でも GC 時に間の不要なオブジェクトが回収され Mark and classified compact と大きな差が生まれなかったということもある．

4.3.2 読み込み回数

実験結果を表 4.3 に示す．読み込み回数が最も少ないのは，ページ置き換えアルゴリズムを LRU にしたものである．次が提案手法で，CFLRU，Mark and compact，LRU の順番に読み込み回数が多くなる．NOR FLash では，読み込みの時間コストと比較して，書き込み時間は読み込みの 10 倍程度で，さらに読み込みの 10^4 倍程度時間の時間を使う消去操作も伴うこともあり，LRU で減らせるキャッシュミスよりも書き込みのオーバーヘッド時間のほうが大きい．犠牲にしたページフォルト数よりも書き込み回数を減らしたことによる時間のほうが大きいという結果であった．

表 4.3. 実験結果

提案手法	45506
Mark and compact	46693
all CFLRU	46402
LRU	39254

第 5 章

まとめと今後の課題

5.1 まとめ

本研究では，搭載しているメモリが少ないスモールデバイスでも，仮想的にメモリを増やすことで容易で効率的な開発ができる言語処理系のための Flash を併用したメモリ管理システムを提案した．書き込みの多いページをできるだけメモリに保つために．reference bit LRU を改良して ページアウトの優先順位を変更するアルゴリズムを提案した．それから，Mark and compact の compaction を改良し，書き込みのあったオブジェクトをまとめることで，クリーンなページだけをページアウトすることも提案した．また，Scheme 処理系をこのメモリ管理システムを用いた仮想機械を実装することで実現した．本手法の有用性を検証するために，ソフトウェアの実行中の Flash 書き込み回数を測定した．書き込み回数は提案手法が最も少なかったが，最適化やプログラムとの相性によってあまり差がでなかった．また，本手法で犠牲にしたページフォルト数についても読み込み数を測定する実験を行った．ページフォルト数を抑えることに向いている LRU が最も読み込み数が少ないという結果であった．Flash メモリでは，読み込み時間よりも書き込み時間のオーバーヘッドが大きく，本手法で犠牲にしたページフォルト数よりも書き込み回数を減らしたことによる時間のほうが大きいという結果であった．

5.2 今後の課題

実験結果から，Flash への書き込みの削減はできるが，不要なオブジェクトが想定以上に発生していることが分かった．これらはアセンブラを CPS 変換したコードを逆変換してから適用したため，クロージャの生成が増えたからである．本システムは，オブジェクトを Flash に割り当てることも考慮しているが，計算に必ずしも必要なものではないため，本手法を用いて，計算に必要なオブジェクトが Flash への書き込み数をどれだけ減らせたのかが分かりにくくなっている．そのため，静的な解析による最適化で不要なクロージャを削減して計算に本当に必要なオブジェクトに関係する Flash への書き込み回数を検証する必要があるだろう．また，不要なコードを削減することは，使える non-volatile 領域が増やすことにつながるため，

実用的なシステムを目指す上でも必要である．また，有用性を示すアプリケーションが少ないため，それらを増やすことでより詳細に有用性を調べる必要もある．

本システムは改良の余地があると考えている．non-volatile 領域を用いるのは計算時間のかかるタスクを想定しており，書き込み数の低減に伴う時間の多少のオーバーヘッドは許容するが，これらを多少向上させることは，時間にもう少しシビアなソフトウェアを適用可能にすることにつながる．一例として，CCF-LRU のように参照頻度をページ置き換えアルゴリズムに組み込むことでページインを減らすことが可能になると考えている．また，volatile 領域がいっぱいになると non-volatile 領域に割り付けるようになる．GC 時に移動する機構は実装しているが，実行前にどこに割り当てるかを予測して，すぐ回収されるようなオブジェクトを優先的に volatile 領域に割り当てるような機構があれば，non-volatile 領域の GC に伴う書き込みが減らせると考えられる．それから，volatile 領域の余裕が少なくなると，volatileGC が頻発してしまうことも実験中のシステムの挙動を確認して分かった．これは，参照カウント以外の GC で起こる問題であり，適切な場所に割り当てることでこの問題も低減できるだろう．確保場所の事前選択は，静的解析や実行時のプロファイルの記録によって，ソースコードにアノテーションするようなことで実現できる．

書き込み耐性の低い記憶デバイスでは，書き込みを分散させることで寿命を伸ばすウェアレベリングという手法が知られている．ウェアレベリングは本システムでは未実装である．これを実装すれば，本システムを採用した場合での Flash の寿命が伸びるため，ウェアレベリングの実装も今後の課題である．

発表文献と研究活動

(1) 井上 曜, 千葉 滋. 不揮発性メモリの性質に特化した書き込み回数を抑えるメモリ管理の提案計, 第 22 回プログラミングおよびプログラミング言語ワークショップ, 2020. (ポスター発表)

(2) 井上 曜, 千葉 滋. 不揮発性メモリの併用に特化した IOT 向けプログラム言語ランタイム 日本ソフトウェア科学会第 37 回大会, (口頭発表)

参考文献

[1] Onur Mutlu Benjamin C. Lee, Engin Ipek and Doug Burger. Architecting phase change memory as a scalable dram alternative. *In Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[2] Onur Mutlu Benjamin C. Lee, Engin Ipek and Doug Burger. Architecting phase change memory as a scalable dram alternative. *InProceedings of the 36th Annual International Symposium on ComputerArchitecture (ISCA)*, 2009.

[3] Kathryn S. McKinley Shoaib Akram, Jennifer B. Sartor and Lieven Eeckhout. Crystal gazer: Profile-driven write-rationing garbage collection for hybrid memories. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2019.

[4] Kathryn S. McKinley Shoaib Akram, Jennifer B. Sartor and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. *ACM SIGPLAN Conference on Programming Language Design andImplementation*, 2018.

[5] X. Renshaw Wang Xinwei Guan Lan Wang Shishen Yan Tom Wu and Xiaolin Wang Qiang Cao, Weiming Lü. Nonvolatile multistates memories for high-density data storage. *ACS Appl. Mater. Inter*, 2020.

[6] Marc Feeley Vincent St-Amour. Picobit: A compact scheme system for microcontrollers. *IFL'09: Proceedings of the 21st international conference on Implementation and application of functional languages*, 2009.

[7] Eliot Moss Richard Jones, Antony Hosking. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman&Hall/CRC, 2011.

[8] H. ieberman and C.E. Hewitt. A real-timegarbage collector based on the lifetimes ofobjects. *Communications of the ACM*, Vol. 26, No. 6, pp. 419–429, 1984.

[9] J.-u. Kang J.-s.Kim S.-y.Park, D.Jung and J.Lee. Cflru. *SIAM J. Alg. Disc. Math.*, pp. 234–241, 2006.

[10] S. Park S. Kang H. Jung, H. Shim and J. Cha. Lru-wsr: Integration of lru and writes sequence reordering for flash memory. *IEEE Trans. Consumer Electron*, Vol. 54, No. 3, pp. 1215–1223, 2008.

[11] X. Su K. Cui L. Yue Z. Li, P. Jin. A new buffer replacement algorithm for flash memory, trans. on cons. *IEEE Transactions on Consumer Electronics*, 2009.

[12] Waite W.M. chorr, H. efficient machine-independent procedure forgarbage collection in various list structures. *Commun. ACM10*, 1967.

[13] L.Chao Z.Li, X.Peng and Z.Xu. Everylite:a lightweight scripting language for micro tasks in iot system. pp. 381–386, 2018.

[14] Ruben Ayrapetyan Andrey Shitov Evgeny Gavrin, Sung-Jae Lee. Ultra lightweight-javascript engine for internet of things. *SPLASH Companion 2015: Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications:Software for Humanity*, 2015.

[15] Kelsey-R. Rees J. Hudak P. Philbin J. Kranz, D. and N Adams. Orbit: an optimizing compiler for scheme. in sigplan '86 symposium on compiler construction. *SIGPLAN '86 Symposium on Compiler Construction*, Vol. 21, No. 7, pp. 219–233, 1986.

謝辞

本研究を進めるにあたり，研究方針の相談や論文・発表についての助言などを指導してくださった千葉 滋教授に心より感謝いたします．研究生活を通じてご支援くださった千葉滋研究室のみなさんにも感謝いたします．

実験に利用したアプリケーションのソースコード

Listing 5.1. 実験に使用したソースコード

```
(import (scheme base) (scheme fixnum))

(define rand-val 3)
(define rand-256
  (lambda ()
    (set! rand-val (fxremainder (fx+ (fx* 19 rand-val) 13) 126))
      rand-val))

(define make-acc-colors
  (lambda (n-colors)
    (let ((res (make-vector n-colors)))
      (let loop ((i (fx- n-colors 1)))
        (let ((_v (vector 0 0 0 0)))
          (vector-set! vec i _v)
          (unless (fx=? i 0) (loop (fx- i 1)))))
      res)))

(define pow (lambda (n) (fx* n n)))

(define color-diff
  (lambda (color1 color2)
    (fx+ (pow (fx- (vector-ref color1 0)
                   (vector-ref color2 0)))
         (fx+ (pow (fx- (vector-ref color1 1)
                        (vector-ref color2 1)))
              (pow (fx- (vector-ref color1 2)
                        (vector-ref color2 2)))))))

(define calc-nearlest-color
  (lambda (color centers)
```

```
31     (let loop ((index 1)
32                (res 0)
33                (min-v (color-diff color (vector-ref centers 0))))
34       (if (fx=? (vector-length centers) index)
35           res
36           (let ((v (color-diff color (vector-ref centers index))))
37             (if (fx<? v min-v)
38               (loop (fx+ index 1) index v)
39               (loop (fx+ index 1) res min-v)))))))
40
41 (define make-acc-colors
42   (lambda (n-colors)
43     (let ((res (make-vector n-colors)))
44       (let loop ((i (fx- n-colors 1)))
45         (let ((_v (make-vector 4)))
46               (vector-set! _v 0 0)
47               (vector-set! _v 1 0)
48               (vector-set! _v 2 0)
49               (vector-set! _v 3 0)
50               (vector-set! vec i _v)
51               (unless (fx=? i 0) (loop (fx- i 1)))))
52       res)))
53
54 (define acc->center!
55   (lambda (center acc)
56     (let loop ((i 0))
57       (unless (fx=? (vector-length acc) i)
58         (let* ((acc-col (vector-ref acc i))
59                (new-color (vector-ref center i)))
60             (unless (fx=? (vector-ref acc-col 3) 0)
61                 (vector-set! new-color 0
62                              (fxquotient (vector-ref acc-col 0)
63                                          (vector-ref acc-col 3)))
64                 (vector-set! new-color 1
65                              (fxquotient (vector-ref acc-col 1)
66                                          (vector-ref acc-col 3)))
67                 (vector-set! new-color 2
68                              (fxquotient (vector-ref acc-col 2)
69                                          (vector-ref acc-col 3))))
70             (loop (fx+ i 1)))))
71     center))
72
73 (define update-vertical!-loop
```

```
  (lambda (vertical x center)
    (unless (fx=? x (vector-length vertical))
      (let ((center-index (vector-ref (vector-ref vertical x) 0)))
        (vector-set! vertical x (vector-ref center center-index))
        (update-vertical!-loop vertical (fx+ x 1) center))))))

(define update-img!
  (lambda (img center)
    (let loop ((y 0))
      (unless (fx=? y (vector-length img))
        (update-vertical!-loop (vector-ref img y) 0 center)
        (loop (fx+ y 1))))))

(define vertical-walk
  (lambda (vertical x acc-colors center last-flag)
    (unless (fx=? (vector-length vertical) x)
      (let loop ((x 0))
        (let* ((near-color-index (calc-nearlest-color (vector-ref vertical
             x) center))
              (acc (vector-ref acc-colors near-color-index))
              (pixel (vector-ref vertical x)))
          (vector-set! acc 3 (fx+ (vector-ref acc 3) 1))
          (vector-set! acc 0 (fx+ (vector-ref acc 0) (vector-ref pixel
              0)))
          (vector-set! acc 1 (fx+ (vector-ref acc 1) (vector-ref pixel
              1)))
          (vector-set! acc 2 (fx+ (vector-ref acc 2) (vector-ref pixel
              2)))
          (when last-flag (vector-set! pixel 0 near-color-index))
          (loop (fx+ x 1)))))))

(define img-walk
  (lambda (img y acc-colors center last-flag)
    (unless (fx=? (vector-length img) y)
      (vertical-walk (vector-ref img y) 0 acc-colors center last-flag)
      (img-walk img (fx+ y 1) acc-colors center last-flag))))

(define k-means
  (lambda (img n-center n-iter)
    (let ((center (make-initial-center n-center)))
      (let loop ((i n-iter)
                 (center center))
        (let ((acc-colors (make-acc-colors n-center)))
```

```
            (img-walk img 0 acc-colors center (fx=? i 1))
            (if (fx=? i 1)
              (update-img! img (acc->center! center acc-colors))
              (loop (fx- i 1) (acc->center! center acc-colors))))))))

(define img
  #(#(#(#(131 135 75) #(112 123 86) #(96 100 84) #(86 100 93) #(94 112
        106) #(113 117 103) #(114 125 103) #(139 148 100))
    #(#(137 139 82) #(131 140 107) #(104 114 113) #(116 109 108)
        #(122 124 125) #(110 131 135) #(126 143 118) #(142 156 110))
    #(#(123 120 79) #(127 130 100) #(117 172 208) #(203 136 139)
        #(196 155 165) #(116 177 214) #(129 153 136) #(135 152 114))
    #(#(110 107 77) #(101 105 88) #(133 185 223) #(214 133 142) #(199
         153 169) #(132 183 218) #(111 134 112) #(106 125 93))
    #(#(164 145 82) #(64 66 53) #(128 164 198) #(208 133 147) #(187
        155 176) #(121 160 193) #(88 106 86) #(89 102 75))
    #(#(198 163 71) #(118 125 109) #(145 105 124) #(226 102 104)
        #(211 117 125) #(108 123 154) #(98 114 110) #(103 108 88))
    #(#(197 167 87) #(163 147 111) #(180 132 131) #(229 89 81) #(233
        88 75) #(141 128 135) #(143 154 142) #(121 132 114))
    #(#(151 157 135) #(143 133 100) #(172 168 152) #(192 146 143)
        #(190 145 139) #(156 151 141) #(132 132 100) #(109 115 105)))
        )

(k-means img 16 10)
```