メモリチップ内で動く Java 仮想機械

一野瀬 知輝[†1,1,a)]　森本 龍[1,b)]　鵜川 始陽[1,c)]

概要：近年，CPU の計算速度とメモリアクセス速度の乖離が性能上のボトルネックになっている．これを克服するため，メモリの近くで計算をする Processing-in-Memory (PIM) アプローチが注目されている．本研究では，PIM の実装の一つである UPMEM で，CPU からオフロードされた Java のメソッドを実行する機構を開発する．UPMEM は DRAM の近くに小規模なプロセッサである DPU を備えている．本研究では，オフロードする Java メソッドだけを DPU の機械語にコンパイルする．DPU のプログラムメモリは 32KB しかないため，複数のバイナリを入れ替えながら実行する．また，DPU が持つハードウェアスレッドを利用して SPMD 方式の並列プログラミングをサポートする．

キーワード：processing-in-memory, Java, scratch-pad memory, garbage collection, accelerator

A Java Virtual Machine for In-Memory Execution

Kazuki Ichinose[†1,1,a)]　Ryu Morimoto[1,b)]　Tomoharu Ugawa[1,c)]

Abstract: Recently, the gap between CPU computation speed and memory access speed has become a major performance bottleneck. To address this problem, the Processing-in-Memory (PIM) approach, which places computation near memory, has attracted attention. In this work, we use one of the practical implementations of PIM, UPMEM, and develop a system that executes Java methods offloaded from the CPU on UPMEM. UPMEM equips small processors called DPUs near DRAM. Our system compiles only the offloaded Java methods into DPU machine code. Since the program memory of a DPU is limited to 32 KB, our system dynamically swaps binary during execution. Furthermore, we leverage the hardware threads available on the DPU to support SPMD-model parallel programming.

Keywords: processing-in-memory, Java, scratch-pad memory, garbage collection, accelerator

1. はじめに

近年，CPU の計算速度とメモリアクセス速度の乖離が性能上のボトルネックになっている．現在の計算機システムでは，データは DRAM に格納されており，計算の際に CPU とメインメモリの間のバスを通じてデータを転送する．しかし，CPU の計算速度の向上に対してメインメモリに使われる DRAM のレイテンシや帯域の改善は遅く，そのギャップは広がり続けている [1]．CPU にはキャッシュメモリが搭載されているが，広範囲なメモリにアクセスするワークロードではキャッシュの効果は限られる．そのため，メモリインテンシブなアプリケーションでは DRAM アクセスの速度が性能上のボトルネックになっている．

これを克服するため，メモリの近くで計算をする Processing-in-Memory (PIM) アプローチが注目されている．このアプローチに基づく計算機では，メインメモリを構成する DRAM チップそれぞれにプロセッサを配置する．これらのプロセッサが，DRAM チップに並列にアクセスすることで，広帯域なメモリアクセスを実現できる．

UPMEM PIM [2] は実世界で利用可能な PIM アプローチのアクセラレータとして登場し，PIM のソフトウェアの研

This is a manuscript for an unrefereed presentation, which should not preclude its subsequent publication.

[1] 東京大学
The University of Tokyo
[†1] 現在，株式会社フィックスターズ
Presently with Fixstars Corporation
[a)] kazuki.ichinose@fixstars.com
[b)] mryuu@g.ecc.u-tokyo.ac.jp
[c)] tugawa@acm.org

究で注目された [3], [4], [5]．本稿執筆現在，UPMEM PIM のサポートは終了している[*1]が，その後も UPMEM PIM を使った研究は行われている [6]．また，メモリを分散配置するメニーコアプロセッサは組込みシステムやハイパフォーマンス計算でしばしば見られる．例えば Kalray Massively Parallel Processor Array（MPPA）[7] は 256 コアのメニーコアプロセッサで，16 コアで 1 ノードを構成する．各ノードはローカルメモリと DMA コントローラを持ち，16 コア全てが同じプログラムを実行する Single Program Multiple Data（SPMD）構成である．また，PEZY-SC4 プロセッサ [8] は各 PE が 24KB のローカルメモリを備える Multiple Instruction Multiple Data（MIMD）構成である．

UPMEM PIM を使った計算機システム（以下，UPMEM システム）は，CPU と普通の DRAM を備えた従来の計算機構成に加え，メインメモリの一部として UPMEM PIM を備えたハイブリッドな構成になる．UPMEM PIM は，プロセッサを搭載した多数の DRAM チップ（以下，**PIM チップ**）で構成されている．アプリケーション全体の制御を CPU と通常の DRAM で行い，データインテンシブな処理を UPMEM PIM の PIM チップにオフロードするような使い方が想定されている．

しかし，UPMEM システムのアプリケーション開発は煩雑であり，抽象度の低いプログラミングが要求される．PIM チップを使うには，そこで動作する機械語命令列をアプリケーションの実行中に CPU から PIM チップにロードする必要がある．この機械語命令列は，一般には CPU で動作するプログラムとは別に PIM チップのプログラムを作成して，機械語命令列にコンパイルすることで準備する．PIM チップのプログラムメモリの容量の制限から，PIM チップのプログラムを機能毎に分けて複数作る場合もある．さらに，PIM チップ向けのコンパイラは C コンパイラしか用意されておらず，CPU 側のプログラムをオブジェクト指向言語で記述したとしても，PIM チップの処理は抽象度の低い C 言語で記述する必要がある．

PIM を用いたアプリケーションの生産性やメモリ安全性を高めるために，我々は CPU で実行する Java プログラムの一部の実行をシームレスに PIM チップにオフロードするフレームワーク「義経」を開発している [9], [10]．義経では，各 PIM チップが独立した Java ヒープを持つ．CPU で実行している Java のプログラムは，オフロードしたい処理を記述したメソッドを持つオブジェクトを PIM チップのヒープに作る．これには義経の API を使う．このとき得られるオブジェクトのハンドルを使ってオフロードしたいメソッドを呼び出すことで，メソッドが PIM チップで実行される．

本研究では，義経フレームワークのために，PIM チップ内で Java メソッドを実行する機構を開発する．本機構は，オフロードする Java メソッドと，そこから呼び出される可能性のあるメソッドを PIM チップ用のバイナリにコンパイルする．このバイナリにはクラス情報の管理や GC などのランタイムシステムも含まれている．オフロードする処理が複数あるときは，処理に対応するメソッド毎にバイナリを作る．

PIM チップ用にメソッドをコンパイルする際には，PIM チップのメモリ構成と並列処理が課題になる．UPMEM PIM の PIM チップのプログラムは 32 KB のプログラムメモリに配置する必要がある．オフロードするメソッドは，ランタイムシステムを加えてもこの容量に収まるようにコンパイルしなければならない．また，高速にアクセスできるデータ用のメモリであるスクラッチパッドメモリも 64 KB しかない．ユーザーデータだけでなく最大 16 個のハードウェアスレッドの実行スタックもここに配置されるため，深い再帰呼出しに備えて実行スタックを大きくするとユーザーデータの領域が小さくなる．PIM チップのプロセッサにとって DRAM は外部装置である．直接ロードストア命令ではアクセスできず，スクラッチパッドメモリとの間で DMA でデータを転送する必要があり，特に細粒度の DRAM アクセスはオーバヘッドが大きい．

UPMEM PIM の PIM チップは 16 個のハードウェアスレッドを備え，Single Program Multiple Data (SPMD) モデルの並列処理をサポートする．DPU の命令パイプラインは一つで，各スレッドは 11 サイクルに 1 回しかパイプラインに命令を供給できない．そのため，パイプラインを完全に埋めるには 11 個のアクティブなスレッドが必要になる．本研究でも，並列処理のサポートは必須である．

本研究では，Java メソッドの関数フレームを実行スタックの外で明示的に管理し，そのうえで全てのメソッド呼出しをトランポリン方式にコンパイルして C 実行スタックを伸ばさないようにする．Java メソッドの関数フレームは，カレントフレーム周辺を除いて DRAM に保存することで，スクラッチパッドメモリを消費せずに深い再帰呼出しに対応する．オフロードされる処理が一つのバイナリに収まらないときは，メソッド単位で分割して，バイナリを切り替えながら実行する．

ガーベージコレクション（GC）には，義経フレームワーク用に開発した世代別 GC である gray-in-young GC [11]（GiY GC）を使う．GiY GC はスクラッチパッドメモリの余った領域を新世代領域，DRAM を旧世代領域とする．オブジェクトの昇格の際には，新世代領域でポインタを更新したあと，1 回の DMA でオブジェクト全体を旧世代にコピーする．また，メジャー GC のプログラムは専用のバイナリに分離する．

並列処理では義経のプログラムでも SPMD モデルの並列処理をサポートし，同じメソッドを最大 16 個のスレッ

[*1] 2025 年 5 月に終了した．

ドで実行する．同期プリミティブにはバリア同期を提供する．複数のスレッドがバイナリの切替えやバリア同期，GC を並行して要求することがあるので，それをスケジューラで調停する．

本研究で開発した Java メソッド実行機構を，いくつかのベンチマークプログラムを使って評価した．実行時間を UPMEM PIM の PIM チップ向けに同じプログラムを C 言語で書いたものと比較したところ，プログラムによってほぼ同じ時間のものから，100 倍程度遅いものまで様々であった．詳細な調査から，DRAM に置かれたオブジェクトに頻繁にアクセスする場合や，頻繁にメソッド呼出しを行う場合に遅くなり，一方で，メモリ割り当ては C 言語より高速であることが分かった．これらの処理をいずれも行わず，ループで計算だけを行うプログラムの実行時間はほぼ変わらなかった．また，GC に要した時間は無視できる程度であった．また，バイナリの入替えには平均 2.74 ms かかった．これは 400 MHz で動作する PIM チップで約 1,000,000 サイクルに相当する．バイナリのサイズについては，C 言語と比べると平均で 6.02 倍であった．バイナリのうち，GC や並列処理，トランポリン呼出しなどの仕組みを実装したランタイムシステムが約 10 KB 使っていた．

本論文の位置付け

本論文は我々の過去に発表した研究の成果を発展させ統合したものである．バイナリを入れ替えながら実行する実行方式（3.3 節）は第 41 回日本ソフトウェア科学会大会 [12] で試作した．第 27 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では設計を変更して並列処理（3.5 節）に対応する実行方式を発表した[*2]．本論文では，これを基に最適化した実装を示す．GC（3.8.1 節）は第 41 回日本ソフトウェア科学会大会 [13] で発表し，International Symposium on Memory Management 2025 [11] で発展させたものを利用した．

2. 研究背景

2.1 UPMEM PIM

PIM は，従来のアーキテクチャにおける CPU とメインメモリの間のデータ転送によるボトルネックを解消するために提案されたアプローチである．従来のアーキテクチャでは，CPU で計算を行うために CPU とメモリの間のバスを通じてデータを転送する必要があり，データインテンシブなアプリケーションにおいてボトルネックになっていた．PIM アプローチはメインメモリを構成する DRAM チップ上や DRAM チップのすぐ近くにプロセッサを配置する．多数のプロセッサが並列に DRAM にアクセスすることで，広帯域なメモリアクセスを実現する．

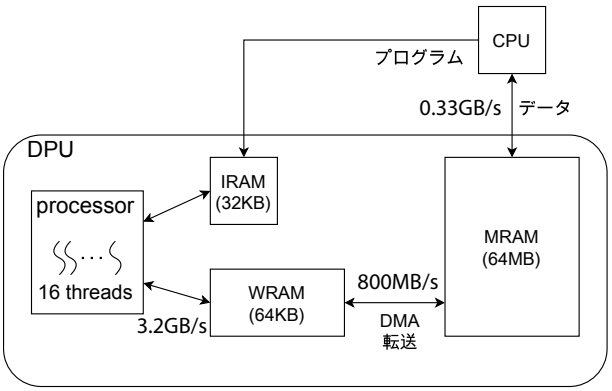


図 1 UPMEM PIM の PIM チップ
Fig. 1 PIM chip of UPMEM PIM.

UPMEM システムは実世界で利用可能な PIM アプローチのアクセラレータを備えた計算機システムである．UPMEM システムは CPU と普通の DRAM に加え，プロセッサを備えた DRAM である UPMEM PIM で構成されたハイブリッドなシステムである．UPMEM PIM には合計で 2560 個の PIM チップが搭載されている．UPMEM PIM もメモリバスで CPU と接続されているが，ソフトウェアからはメインメモリとしては扱えず，UPMEM API を使って PIM チップを制御する．

UPMEM PIM の PIM チップは DRAM Processing Unit（DPU）と呼ばれ，**図 1** のように構成されている．DPU は演算装置，プログラムメモリ（IRAM），Working RAM（WRAM） と呼ばれるスクラッチパッドメモリ（SPM），および DRAM であるメインメモリ（MRAM）で構成されている．MRAM 上のデータは演算装置から直接アクセスすることはできず，DMA を使って WRAM との間でデータを転送する．図中のスループットは Gómez-Luna らによる実測値 [3] を元に，本研究で用いた DPU の動作クロックに合わせて計算したものである．

DPU は 16 個のハードウェアスレッドを備える．これらのスレッドを使って Single Program Multiple Data (SPMD) モデルの並列処理が可能である．つまり，全てのスレッドは IRAM にロードされている同じバイナリを実行するが，独立したプログラムカウンタを持つ．DPU の命令パイプラインは一つで，各スレッドは 11 サイクルに 1 回パイプラインに命令を供給できる [3]．したがって，パイプラインを完全に埋めるには 11 個のアクティブなスレッドが必要である．ただし，DMA コントローラは一つしかなく，ボトルネックになり得る．**図 2** と**図 3** に転送サイズごとの DMA のスループットの実測結果を示す．横軸は同時に転送するスレッド数で縦軸に合計の転送レートを示す．転送サイズが 1024 バイトでは 1 スレッドでほぼ最大スループットになっている．転送サイズが 8 バイトの時はループの制御や

[*2] 一野瀬 知輝, 鵜川 始陽: メモリチップ内で動く Java 仮想機械の試作，第 27 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, 2025.

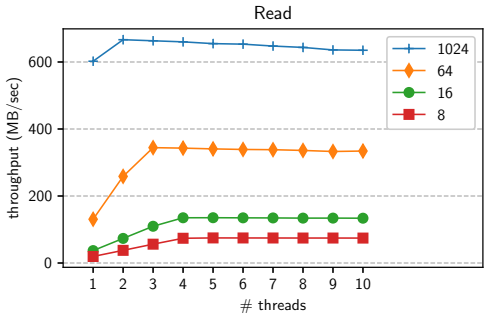


図 2 DMA スループット（リード）
Fig. 2 DMA throughput (read)

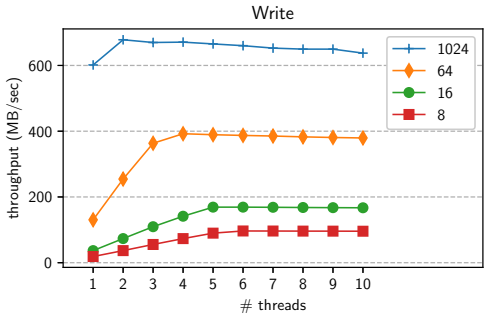


図 3 DMA スループット（ライト）
Fig. 3 DMA throughput (write)

DMA のためのレジスタの設定などの DMA コントローラを使わない時間の割合が大きくなるが，それでもリードで４スレッド，ライトで５スレッド程度で最大スループットに到達している．

2.2 UPMEM のアプリケーション開発

UPMEM の CPU 側のソフトウェアは UPMEM API を使って DPU を制御する．DPU に実行をオフロードする典型的な手順は次のようになる．

(1) 対象の DPU の IRAM にあらかじめ作っておいたバイナリをロードする．
(2) CPU に接続された普通の DRAM から対象の DPU の MRAM に入力データを転送する．
(3) DPU を起動する．
(4) MRAM から普通の DRAM に実行結果を転送する．

何回も同じバイナリを実行するときは，2 回目以降はバイナリをロードする必要はない．バイナリをロードすると WRAM の内容はクリアされるが，MRAM の内容は保存される．これを利用して，データベースのデータを MRAM に配置し，クエリの種類毎に異なるバイナリを実行するというような使い方 [14] ができる．

DPU のプログラムは C 言語で記述する．C 言語上からは WRAM のアドレスは普通のポインタで扱い，MRAM のアドレスは`__mram_ptr` の修飾が付いた **MRAM ポインタ**型で扱う．WRAM も MRAM も 0 番地から始まるので，型がなければ WRAM と MRAM のアドレスは区別できない．WRAM と MRAM はアクセスする方法が異なるので，C コンパイラはポインタの型によって適切な命令列を生成する．具体的には，WRAM にはロードストア命令でアクセスするが，MRAM ポインタを使ったメモリアクセスには DMA 転送を伴う命令列を生成する．これとは別に，DMA コントローラを直接制御して最大 2 KB までのデータブロックを転送する API が利用できる．

```
move r0, x
add r1, id8, __sw_cache_buffer
ldma r1, r0, 0
and r0, r0, 4
add r0, r1, r0
lw r0, r0, 0
```

図 4 MRAM の変数 x の読出し
Fig. 4 Read from variable x in MRAM.

MRAM ポインタを使った MRAM アクセスには長い命令列を要する．**図 4** に MRAM の変数 x の内容を 32 ビットのレジスタ r0 に読み出す命令列を示す．1 行目は DMA の転送元の設定である．2 行目では WRAM 上のバッファである`__sw_cache_buffer` のうち，このスレッドが使うバッファ領域を転送先に設定している．3 行目で DMA の最小単位である 8 バイトを転送している．4, 5 行目は転送先のバッファのうち，変数 x に相当する部分のアドレスを計算する命令であり，6 行目でそのアドレスから r0 に読み出している．MRAM に 32 ビットの書込みをする場合，read-modify-write を行う必要があり，さらに長い命令列が必要になる．

WRAM のメモリ管理のためには buddy system を実装したメモリアロケータが提供されている．一方，MRAM を管理するメモリマネージャはベンダの SDK では提供されていない．

DPU のプログラムをコンパイルする時，使用するスレッド数や各スレッドのスタックサイズを指定する．リンカはこれに基づき WRAM のレイアウトを決定する．並列処理のためには，自身のスレッド番号を得る API とバリア同期の API が提供されている．

2.3 義経フレームワーク

UPMEM システムのアプリケーションの生産性とメモリ安全性を向上させるために，我々は CPU で動作する Java プログラムの一部を DPU にオフロードするフレームワーク「義経」を開発している [9], [10]．本研究では，このフレームワークのための DPU でメソッドを実行する機構を開発する．ここでは，本研究の前提となる義経フレームワークの概要を述べる．

義経フレームワークでは分散オブジェクトモデルに基づいてプログラムを記述する．このモデルでは，CPU と各

```
@DPUClass
class NBodySystem {
  void advance(float delta) {...}
  ...
}

IDPUProxyObject handle =
   UPMEM.createObject(0, NBodySystem.class);
NBodySystem nb = new NBodySystemProxy(handle);
nb.advance(0.01);
```

図 5 義経フレームワークのプログラム例
Fig. 5 Example program of Yoshitsune framework.

DPU はメモリを共有しない計算ノードである．各ノードは独立したヒープを持つ．プログラムの実行は CPU で始まる．CPU から DPU への通信は遠隔メソッド呼出しにより行われる．このとき，引数や戻り値はプリミティブか，DPU 内のオブジェクトのハンドルに限られる．プログラム中では，ハンドルをそのオブジェクトのクラスのサブクラスとして定義したプロキシに包んで扱う．また，CPU から DPU 内のヒープにオブジェクトを生成したり，配列を転送することができる．DPU から CPU や DPU 間での通信はできない．

各オブジェクトはいずれかの計算ノードに配置され，メソッドの実行はそのオブジェクトがある計算ノードで実行される．具体的な記述例として，N 体問題シミュレーションのプログラムの一部を**図 5** に示す．N 体問題を表すクラスは`@DPUClass` のアノテーションが付いた `NBodySystem` である．ユーザーはフレームワークが提供する API である `createObject` を呼び出すことで `NBodySystem` クラスのインスタンスを MRAM に作成する．第一引数の 0 はインスタンスを生成する DPU の番号である．その返り値の `IDPUProxyObject` 型の値は，MRAM 内のインスタンスのハンドルである．これを，`NBodySystem` のサブクラスのプロキシで包む．ユーザーは，`NBodySystem` クラスに定義された `advance` を呼び出すことで，そのメソッドをインスタンスが配置された DPU で実行できる．

3. PIM チップでの Java メソッドの実行

3.1 基本的な実行の仕組み

本研究で開発する Java メソッド実行機構は，メソッドを DPU の機械語にコンパイルするコンパイラと，DPU 側のランタイムシステム，および DPU の実行を制御する CPU 側のランタイムシステムからなる．コンパイラは PIM チップにオフロードされるメソッド，およびそこから呼び出されるメソッドをバイナリにコンパイルする．CPU 側から直接呼び出されるメソッドを**入口メソッド**と呼ぶ．このコンパイラは，入口メソッド毎にバイナリを生成する．

CPU で実行中のユーザープログラムから DPU にあるオブジェクトの入口メソッドが呼び出されると，CPU 側のランタイムシステムは，そのメソッドを実行するためのバイナリを DPU にロードし，実行する．**図 6** にシステムの概観を示す．図 5 の例では `advance` メソッドを呼び出したとき，MRAM には `createObject` で作られたオブジェクトが配置されていて，CPU はそれへのプロキシを持っている．CPU のプログラムが `advance` メソッドを呼び出すと，CPU 側のランタイムシステムは `advance` メソッドがコンパイルされているバイナリを IRAM にロードし，DPU を起動する．DPU では指定された数のハードウェアスレッドで `advance` メソッドを実行する．ハードウェアスレッド数は `advance` のアノテーションで指定する．このバイナリは，全てのスレッドが `advance` メソッドからリターンすると実行が終わるようにコンパイルされる．バイナリの実行が終わると CPU に制御が戻り，CPU 側で `advance` を呼び出したメソッドの実行が再開する．

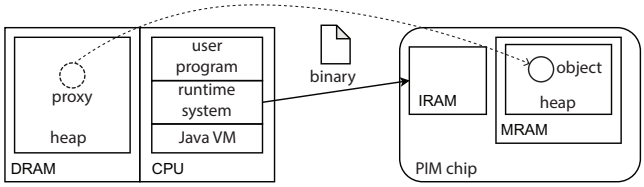


図 6 本機構の概観
Fig. 6 Overview of the system.

3.1.1 ヒープの配置

本機構では世代別 GC でヒープを管理し，新世代領域を WRAM に，旧世代領域を MRAM に配置する．作られて間もないオブジェクトはアクセスされる可能性が高いため，高速な WRAM に新世代領域を配置する．スクラッチパッドメモリを備えた CPU で世代別 GC を行うときは，このような配置は一般的である [15], [16].

3.1.2 コンパイル

本コンパイラは Java バイトコード命令列を C 言語にコンパイルする．それを DPU 用の C コンパイラでバイナリにコンパイルする．このとき，一つのメソッドが一つの C 言語の関数になるようにコンパイルする．

メソッドの中は，バイトコード命令を一つずつ対応する C 言語のコード片に変換するようにコンパイルする．このとき，定数プールの参照などは解決し，定数を直接 C 言語のコードに埋め込む．

3.1.3 プログラムオーバーレイ

入口メソッドから呼び出される可能性があるメソッドが多いと，バイナリが 32 KB の IRAM に収まらない．このときは，メソッド単位で分割し，複数のバイナリにコンパイルし，バイナリを入れ替えながら実行する．またメジャー GC も，実行頻度が低いので別のバイナリにコンパイルし，必要なときにバイナリを入れ替えて実行する．

DPU でメソッドを呼び出すとき，呼出し先のメソッドが同じバイナリになければ，継続を保存して CPU に制御

を戻す．CPU はバイナリを入れ替えて DPU を起動することで，呼出し先のメソッドを実行する．その後，また CPU に戻ってバイナリを戻し，継続を実行する．

3.2 Java プログラムの継続

Java プログラムの継続を表すデータは，メソッド呼出しの履歴と，その履歴に含まれる各呼出し毎に Java のローカル変数，オペランドスタック，およびプログラムの実行位置の情報で構成される．プログラムの実行位置については，バイナリを入れ替える可能性のある全ての位置にラベルを付け，ラベルに対応する整数をプログラムカウンタの代わりに用いる．以降では，この整数を**エントリポイント識別子**と呼ぶ．中断したメソッドの実行を再開するときは，メソッドをコンパイルした関数の先頭で保存しておいたエントリポイント識別子に対応するラベルにジャンプする．これは，Java のプログラムのマイグレーション機構である JavaGo[17] がプログラムの実行位置を保存するのに使っているのと同じ手法である．

保存したエントリポイント識別子や，各メソッドのローカル変数とオペランドスタックは，**Java スタック**と呼ぶデータ構造に保存し，明示的に管理する．Java スタックはバイナリを入れ替えても内容が維持される MRAM に配置する．ただし，MRAM のアクセスは遅く，アクセスのための命令列も長くなるのでスタックトップの数フレームは WRAM 中の固定長領域に作っておき，バイナリを入れ替える時に MRAM に移動させる．WRAM 中の Java スタック領域は各スレッド 128 バイトであり，16 スレッドで 2 KB になる．また WRAM 中のスタックがオーバフローすると WRAM 中のフレームを全て MRAM に移動させ，アンダーフローすると MRAM から 1 フレームだけ WRAM に移動させる．

3.3 メソッドの呼出し

3.3.1 トランポリン方式

C 言語の実行スタックは WRAM に配置される．WRAM の容量は限られており，また，WRAM の余った領域はヒープの新世代領域として利用する．そのため，実行スタックに割り当てる WRAM の領域はできるだけ小さくしたい．一方で，各メソッドを C 言語の関数にコンパイルしているので，再帰呼出しをするプログラムを普通にコンパイルすると，実行スタックが大きくなる．

本研究では，再帰呼出しの深さによらず実行スタックに割り当てる領域を固定長にするために，メソッド呼出しはトランポリン方式の関数呼出しにコンパイルする．つまり，メソッドを呼び出すとき，直接呼び出すのではなく呼び出したいメソッドの識別子を戻り値にして `main` 関数に戻る．以降では `main` 関数を**スケジューラ関数**と呼ぶ．スケジューラ関数は識別子に対応するメソッドを呼び出す．呼び出されたメソッドからリターンする時は，Java スタックに保存された戻り先メソッドの識別子を読み出して，それを戻り値にスケジューラ関数に戻る．

図 7(a) はトランポリンを利用しなかった場合の実行スタックの様子である．再帰的に二分探索をする `binSearch` メソッドが呼び出されると，実行スタックがそれに伴って伸びる．図 7(b) はトランポリンを利用した場合の実行スタックと Java スタックの様子である．再帰的に `binSearch` を呼び出すときに一旦スケジューラ関数にリターンしてから，スケジューラ関数が `binSearch` を呼び出す．このようにすることで，実行スタックにはスケジューラ関数のフレームと現在実行中のメソッドをコンパイルした関数のフレームの二つしか積まれず，再帰呼出しが深くなっても問題ない．

3.3.2 バイナリの入れ替えを伴う場合

メソッドを呼び出すとき，呼出し側は現在実行中のメソッドの識別子と，実行位置を表すエントリポイント識別子を Java スタックに積む．そのうえで，レシーバオブジェクトのクラス情報内にある仮想関数表を使って得た，呼出し先メソッドの識別子を戻り値とし，スケジューラ関数に戻る．スケジューラ関数は，呼出し先メソッドが同じバイナリにある場合，それを呼び出す．同じバイナリに存在しなければ，CPU に制御を返す．CPU 側のランタイムシステムは，MRAM に保存された Java スタックから呼出し先メソッドを特定し，それを含むバイナリをロードして，再度 DPU を起動する．

リターンの時は戻り値を Java スタックに格納した後，リターン先のメソッドの識別子を Java スタックから取り出して，それを戻り値としてスケジューラ関数に戻る．スケジューラ関数はそのメソッドを呼び出す．リターン先メソッドの先頭では，Java スタックからエントリポイント識別子を取り出し，対応するラベルにジャンプする．リターン先メソッドが同じバイナリになければ，スケジューラ関数は CPU に制御を戻してバイナリを入れ替える．

3.4 マシンレジスタの利用

Proebstin ら [18] は，Java のメソッドを C 言語にコンパイルするとき，Java のローカル変数やオペランドスタックのエントリに対応した C 言語のローカル変数を作る手法を提案している．C 言語のローカル変数は C コンパイラによりマシンレジスタに割り当てられる．

本研究でもこの手法を採用し，実行中のメソッドのローカル変数とオペランドスタックの各エントリは C 言語のローカル変数にキャッシュする．これにより，WRAM にある Java スタックのアクセスを減らすことができ，高速化だけでなく，ロードストア命令の削減によるバイナリサイズ削減の効果が期待できる．

ただし，本システムではメソッド呼出しにトランポリン

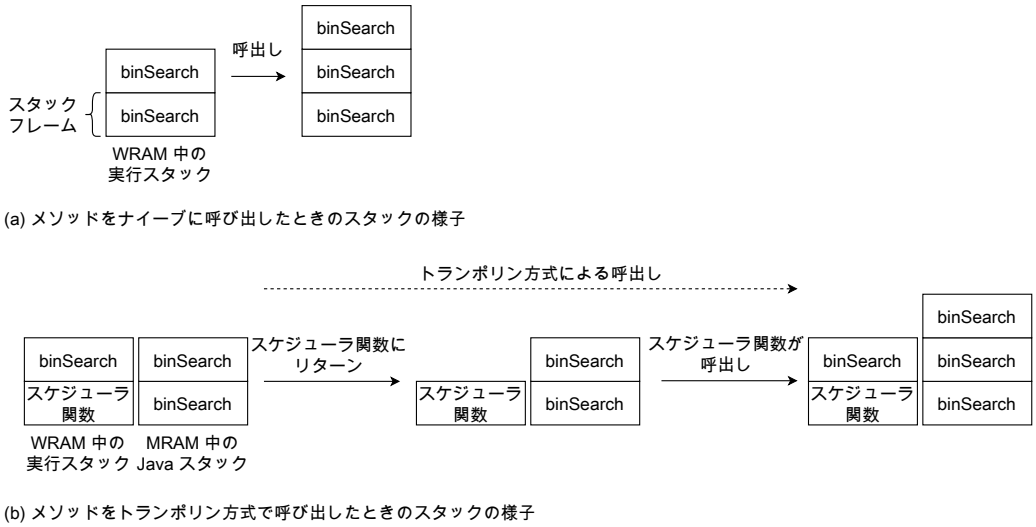


図 7 トランポリン方式のメソッド呼出し
Fig. 7 Method call using the trampoline technique.

を使うので，メソッド呼出し時のレジスタの退避は C コンパイラに頼らず，明示的に行う必要がある．そこで，生存変数解析を行い，メソッド呼出しの前に生存しているローカル変数やオペランドスタックのエントリを WRAM の Java スタックに退避する．一方，復元は関数の先頭の共通処理で行うことで，プログラムを小さく保つ．このときは，いずれかのメソッド呼出しの位置で生存している変数を全て読み出す．

3.5 並列処理

3.5.1 SPMD プログラミング

本システムは DPU が持つ複数のハードウェアスレッドを活用するために，SPMD プログラミングをサポートする．そのために，以下の仕組みを提供する．

- 実行に使うスレッド数の指定
  入口メソッドに付けるアノテーション`@NrTasklets(`n`)`で，そのメソッドを実行するときのスレッド数 n を指定する．CPU から入口メソッドが呼び出されると，指定した数のスレッドが同じ引数で同じメソッドの実行を始める．
- 自身のスレッド番号を得る組込みメソッド
  自身のスレッド番号を得る組込みメソッド `me()` を提供する．このメソッドは DPU 上で実行するときだけ意味のある値を返し，0 から 15 のスレッド番号を返す．
- バリア同期
  バリア同期のための組込みメソッド `barrier_wait()` を提供する．`barrier_wait()` を呼んだスレッドは，他の全てのスレッドが `barrier_wait()` を呼ぶまで待機する．現在は一部のスレッドだけが参加するバリア同期はサポートしていない．このメソッドは CPU 上で呼び出しても何も起こらない．

```
@NrTasklets(3)
void f() {
  switch (me()) {
    case 0: g1(); break;
    case 1: g2(); break;
    case 2: g3(); break;
  }
}
```

図 8 スレッド毎に異なるバイナリを要求するプログラム
Fig. 8 Program where each thread requires different binary.

アノテーションで指定されたスレッド数は，DPU のバイナリを作る際に C コンパイラに渡される．自身のスレッド番号を得る組込みメソッドも UPMEM SDK で提供されている関数を呼び出すことで実現する．バリア同期には UPMEM SDK で提供されているバリア同期の機構をそのまま使うことはできない．バリア同期の実現は 3.6 節で述べる．

3.5.2 バイナリの入れ替え

並列実行すると，各スレッドが異なるタイミングで異なるバイナリを要求する可能性がある．例えば，**図 8** に示すプログラムでは，3 スレッドでメソッド `f` を実行する．その中で，スレッド 0, 1, 2 はそれぞれメソッド `g1`, `g2`, `g3` を呼び出す．このとき，メソッド `g1` だけが `f` と同じバイナリにコンパイルされており，`g2` と `g3` はそれぞれ異なるバイナリにコンパイルされているということがあり得る．

本研究では，実行できるスレッドがある限り現在のバイナリで実行を続ける方針をとる．全てのスレッドが実行を完了した状態（**完了状態**）か，別のバイナリのメソッドを呼び出したり，別のバイナリのメソッドにリターンしようとして待っている状態（**入替待機状態**）になったときにバイナリを入れ替える．スレッドの状態はバイナリを入れ替

えても維持されるよう，MRAM に保存する．

全てのスレッドが完了状態か入替待機状態になると，CPU に制御を戻す．CPU 側のランタイムシステムは要求されたメソッドがコンパイルされたバイナリをロードして実行する．複数のスレッドが異なるバイナリのメソッドを要求するときは，そのうち一つを選んでロードする．

DPU を起動すると，全てのスレッドがスケジューラ関数の先頭から実行を開始する．その先頭で，スレッドが入替待機状態かどうかを調べる．入替待機状態であれば，**実行可能状態**になりメソッドを実行する．ただし，現在のバイナリに実行すべきメソッドがなければ，また入替待機状態に戻る．

3.6 スケジューラ関数

あるスレッドがバリア同期で待機している間に，別のスレッドがバイナリの入れ替えのために入替待機状態でスケジューラ関数を抜けることがある．もし，ナイーブに SDK で提供されたバリア同期を使っていれば，そのバリア同期は完了することがなくデッドロックしてしまう．

また，本研究ではミューテータスレッドも GC に参加して並列に GC する．これは DPU のハードウェアスレッド数に上限があるからである．GC が起動すると，バリア同期で待機しているスレッドや，完了状態や入替待機状態になっているスレッドも GC に参加しなければならない．

本研究では，バリア同期で待機している状態（**同期待機状態**）のスレッドがあるときも，それ以外のスレッドが全て完了状態か入替待機状態であれば，バイナリを入れ替える．また，完了状態や入替待機状態のスレッドも GC になれば動けるよう，スレッドの実行は終了させず，待機させておく．

基本的な方針として，実行を続けられなくなったスレッドは，継続を保存してスケジューラ関数に制御を戻し，状態を適切に変化させて待機する．したがって，バリア同期でも一旦スケジューラ関数に戻る．メモリ不足で GC を起動するときもスケジューラ関数に戻り，**GC 待機状態**になって他のスレッドがセーフポイントに到達するのを待つ．セーフポイントに到達したスレッドもスケジューラに戻り GC 待機状態になる．

最後に待機するスレッドは，他の全てスレッドの状態を調べて，次の動作を決め，全てのスレッドの待機を解除する．以下を順に確認して最初に該当する動作とする．

(1) GC 待機状態のスレッドがあれば GC をする．
(2) 入替待機状態のスレッドがあればバイナリを入れ替える．
(3) 同期待機状態のスレッドがあればそのスレッドの実行を再開する．
(4) 実行を完了する．

同期待機スレッドがメソッドの実行を再開しようとしたとき，バイナリが入れ替わっていることがある．そのときは，すぐに入替待機状態となり，元のバイナリに戻るのを待つ．

3.7 オブジェクト

3.7.1 アラインメント

オブジェクトは DMA の制約に合わせて 8 バイトアラインメントして配置する．さらに，スカラオブジェクトのフィールドは型によらず 8 バイトとする．DMA の最小転送サイズは 8 バイトなので，MRAM にあるオブジェクトのフィールドに書き込むとき，read-modify-write を避けるためである．配列の要素は領域を節約するために，要素の型に合わせたサイズにする．

3.7.2 クラス情報

クラス情報は頻繁にアクセスされるため，WRAM に格納しオブジェクトヘッダに格納された**クラス識別子**をキーとしてアクセスできるようにする．クラス情報にアクセスするのは，メソッド呼出しの際の仮想関数テーブルのアクセス，`instanceof` や参照型配列への代入の際のサブクラス関係の実行時チェック，オブジェクトの生成，および GC である．

WRAM の内容はバイナリをロードする度にクリアされる．そこで，アプリケーションの初期化時にクラス情報を MRAM に配置し，その後，DPU を起動した時に毎回，スケジューラ関数の先頭で MRAM から WRAM にコピーする．WRAM 中のクラス情報のアドレスはバイナリ毎に異なるので，オブジェクトヘッダにはポインタではなく，クラス識別子の整数を格納し，それをインデックスとする配列を使ってクラス情報を得る．

3.7.3 参照

DPU 用の C コンパイラはポインタの型を使って WRAM と MRAM のポインタを区別し，異なる方法でアクセスする（2.2 節参照）．しかし，オブジェクトがどちらにあるか分からない状況では参照の型で領域を区別することはできない．

本研究では，アドレスの最下位ビットに WRAM か MRAM かを表すタグを付けた `uintptr_t` 型の値でオブジェクトの参照を表す．オブジェクトへのアクセスのコードは**図 9** に示す疑似コードのようになる．このコードは，オブジェクト `obj` のオフセット `offset` にあるフィールドを読み出すコードである．このように，タグによって適切な型にキャストしてからアクセスする．これをコンパイルすると長い機械語命令列になるので，バイナリサイズを抑えるため，オブジェクトへのアクセスはランタイムの関数にする．

3.8 ヒープ管理

新世代領域を WRAM に旧世代領域を MRAM に置き，

```
uint64_t get_field(uintptr_t obj, int offset) {
  if ((o & 1)) /* WRAM */
    return ((uint64_t*)(obj & ~1))[offset];
  else /* MRAM */
    /* this access involves DMA */
    return ((__mram_ptr uint64_t*) obj)[offset];
}
```

図 9 オブジェクトのフィールドを読み出す処理の疑似コード
Fig. 9 Pseudo code for reading from a field of an object.

毎回のマイナー GC で新世代領域で生き残ったオブジェクトを全て昇格させる．マイナー GC は全スレッドで並列に行う．各入口メソッドに対応するバイナリには頻繁に起こるマイナー GC だけリンクし，メジャー GC は別のバイナリに分ける．

バイナリを入れ替えるときには，CPU に戻る前にマイナー GC を行い，新世代領域を空にする．これは，バイナリを入れ替えると WRAM の内容がクリアされるからである．

3.8.1 マイナー GC

マイナー GC には我々が本システムのために開発した世代別 GC である gray-in-young (GiY) GC [11] を使う．以下では，GiY GC の概要を述べる．なお，本研究では文献 [11] で提案したうちの OLI キャッシュは採用していない．

昇格の際，Cheney のアルゴリズム [19] のような一般的なコピー GC のアルゴリズムでは，まずオブジェクトを to 空間である旧世代領域にコピーし，その後，コピーしたオブジェクトの中のポインタを探して子オブジェクトをたどったり，子オブジェクトの移動先を指すように更新したりする．これらの処理は，多数のワード単位の旧世代領域のアクセスを伴う．旧世代は MRAM にあり，ワード単位のアクセスはオーバヘッドが大きい．

GiY GC は，子オブジェクトの探索を完了して子オブジェクトへのポインタの更新を終えるまでの間，つまり，Dijkstra の三色の抽象化 [20] でオブジェクトが灰色である間，オブジェクトを新世代領域に留めるところに特徴がある．これらの処理が終わり，オブジェクトが黒になると，1 回の DMA でそのオブジェクト全体をコピーする．GiY GC はオブジェクトグラフを深さ優先探索する．そのときに訪問したオブジェクトには，旧世代領域のコピー先を予約する．この予約した領域を指すフォワーディングポインタを新世代領域のオブジェクトに書き込む．これによりコピー前にオブジェクトの移動先が決まるので，新世代領域で子オブジェクトへのポインタを更新できる．

並列化では，探索のためのスタックを共有することでロードバランシングを行う．各スレッドは，オブジェクトの持つポインタを探すとき，まだフォワーディングポインタが設定されていない子オブジェクトを見つけると，フォワーディングポインタを設定しマークスタックに積む．その後，親オブジェクトの次のポインタを探す．マークスタックに積まれたオブジェクトは，手が空いたスレッドが取り出し，そのオブジェクトの中のポインタを探す．共有マークスタックのアクセスに伴う同期オーバヘッドを抑えるため，各スレッドはマークスタックの一部をキャッシュする．マークスタックは固定長の領域を WRAM に確保しておき，オーバーフローすると MRAM に書き出す．

```
def copy(r)
    // prepare
    if r.forwarding != null
        return r.forwarding
    r.forwarding = alloc()
    stack.push(r)
    // DFS traversal
    while !stack.empty()
        o = stack.pop()
        foreach f in Pointers(o)
            p = o[f]
            if in_young(p)
                if p.forwarding == null
                    p.forwarding = alloc()
                    stack.push(p)
                o[f] = p.forwarding
        dma_copy(o, o.forwarding, o.size)
    return r.forwarding
```

図 10 GiY GC のコピー関数（[11] より引用）
Fig. 10 Copying function of the GiY GC (adapted from [11])

図 10 に GiY GC のコピー関数の疑似コードを示す．引数でルートオブジェクト `r` を受け取ると，`alloc()` 関数でそのコピー先を予約し，フォワーディングポインタを設定して，マークスタックに積む．その後，マークスタックから取り出したオブジェクト `o` の中のポインタを探し，子オブジェクト `p` が見つかると，そのコピー先を予約し子オブジェクトはマークスタックに積む．親オブジェクトのフィールドは，予約した領域を指すように書き換える．全てのポインタを更新し終わると `dma_copy` 関数でオブジェクト `o` 全体をコピーする．

3.8.2 メモリ割当て

メモリ割当ては，各スレッドが Thread Local Allocation Buffer （TLAB）を持ち，バンプポインタ割当てを行う．新世代領域を 1024 バイトで区切り，スレッドはそのどれか一つを TLAB として持つ．TLAB にはそのスレッドだけがメモリを割り当てることができるので，同期は必要ない．TLAB が一杯になると，新しいブロックを TLAB にする．未使用ブロックがなければ，マイナー GC を起動する．

TLAB の半分の大きさである 512 バイト以上のオブジェクトは直接旧世代領域に割り当てる．スカラオブジェクトは通常は 512 バイトに収まるが，配列は要素数によっては 512 バイト以上になる．

3.8.3 CPU からのメモリ割当て

CPU 側ランタイムシステムで DPU にオブジェクトを作ったり，CPU にあるプリミティブ型要素の配列を DPU にコピーできる．これらは MRAM にある旧世代領域に生成する．新世代領域は WRAM にあり，そこに生成しても次のバイナリをロードしたときに消えてしまうからである．

旧世代領域にオブジェクトを生成するとき，旧世代領域が一杯になれば CPU 側からメジャー GC を起動すればよいが，CPU 側からのメジャー GC の起動は未実装である．

4. 評価

本機構を実行速度，バイナリサイズ，バイナリの入替えにかかる時間，および並列実行時のスケーラビリティの観点から評価した．

評価のために Are We Fast Yet（AWFY）ベンチマーク集 [21] の中から小規模なプログラムを義経フレームワークと C 言語に移植した．以下では，それぞれ**義経実装**と **C 実装**と呼ぶ．AWFY ベンチマーク集は同じベンチマークプログラムを様々な言語で実装している．Java 実装から移植した義経実装はプログラム全体を DPU で実行し，バイナリの入替えは起こさない．また，C++実装から移植した C 実装では，データは全て WRAM に作るようにし，動的メモリ確保には UPMEM SDK が提供する buddy system アロケータを使った．このアロケータはスレッドセーフである．

移植した 8 個のベンチマークプログラムに nbody と gemv を加えたベンチマークプログラムの一覧を**表 1** に示す．移植したうちの storage は 4 分木を作るプログラムである．AWFY ベンチマークでは高さ 7 の木を作るが，移植では C 言語版で全てのデータが WRAM に収まる上限の高さ 5 に変更した．

追加したプログラムの一つである N 体シミュレーションの nbody は AWFY ベンチマークを元にしているが，義経フレームワークのために調整している．AWFY ベンチマークからの変更点は以下の 3 点である．

- DPU ではハードウェアで浮動小数点数演算をサポートせず，特に倍精度浮動小数点数型の演算はバイナリが大きくなるので，座標は単精度浮動小数点数型にした．
- 質点オブジェクトのアクセサをインラインに展開してメソッド呼出しを伴わないようにした．
- 平方根を求めるライブラリはないので，ニュートン法で平方根を求めた．

行列とベクトルの乗算をする gemv は並列実行時のスケーラビリティの評価のために実装した．行列の大きさは分割して全 DPU で並列に計算するときの 1DPU 分を想定して 24 行 16384 列とした．行列とベクトルの要素は `int` 型とした．

表 1 に義経フレームワークのプログラムでオーバーヘッドの原因として考えられるイベントの回数を，ベンチマークプログラム毎に示す．項目は MRAM にあるオブジェクトの頻度（obj），メソッド呼出しの頻度（call），Java スタックの WRAM 領域のオーバーフローとアンダーフローの頻度（stack），マイナー GC の回数（GC），および実行時間（time）である．obj と stack は MRAM のアクセスを伴う．call も，レシーバオブジェクトが MRAM にある場合はクラス識別子の読出しに MRAM のアクセスを伴う．また，call ではトランポリン方式の呼出しオーバヘッドも考えられる．

表 1 ベンチマーク
Table 1 Benchmark programs.

benchmark	obj [$\times 10^3$/s]	call [$\times 10^3$/s]	stack [$\times 10^3$/s]	GC [×1]	time [ms]
bounce	65.3	104.5	0.1	3	994.6
list	20.8	161.6	7.2	1	1360.7
mandelbrot	0.0	0.0	0.0	0	1431.3
permute	393.1	96.5	18.4	1	1943.1
queens	299.8	102.0	9.7	2	1185.3
sieve	829.9	0.1	0.0	1	1269.5
storage	171.0	73.9	25.2	195	2442.8
towers	146.1	131.1	37.5	1	2805.3
nbody	0.4	0.2	0.0	0	538.5
gemv	896.0	0.0	0.0	0	2633.3

実験に用いた DPU は UPMEM バージョン v1b のものであり，400 MHz で動作する．本実験では DPU を一つだけ用いる．DPU の詳細は 2.1 節で述べた通りである．UPMEM SDK はバージョン 2025.1.0 で，C コンパイラのバージョンは clang 12.0.0 である．義経実装のコンパイルではバイナリサイズに最適化する `-Os` フラグを，C 実装のコンパイルでは実行時間に最適化する `-O2` フラグを付けた．実験は 4.4 節のスケーラビリティの評価を除いて全てシングルスレッドで行い，新世代領域のサイズは 28 KB とした．旧世代領域は 10 MB としたが，メジャー GC は発生しなかった．ホスト側計算機は CPU が Intel Xeon Silver 4216 CPU @ 2.10GHz × 2，メモリは DDR4-2666 256GB の構成である．OS は Linux Ubuntu 22.04.5 LTS を用いた．ただし，本実験の実行はバイナリの入替え（4.3 節）を除いて全て DPU で行われる．

4.1 実行時間

義経実装の実行時間を**図 11** に示す．また，実行時間が C 実装に近かったプログラムについて，拡大したものを**図 12** 示す．いずれも C 実装に対する比である．

いくつかのプログラムでは C 実装に対して 10 倍以上，towers では 100 倍程度の実行時間がかかっている．これらは，MRAM 中のオブジェクトに対するアクセスの頻度が高いか，メソッド呼出しの頻度が高いか，その両方である．bounce と list はメソッド呼出しの頻度が高い．特に list はアクセサのメソッド呼出しが多く，これらは C 実装ではコンパイラがインライン展開するため，C 実装との比が大きくなった．

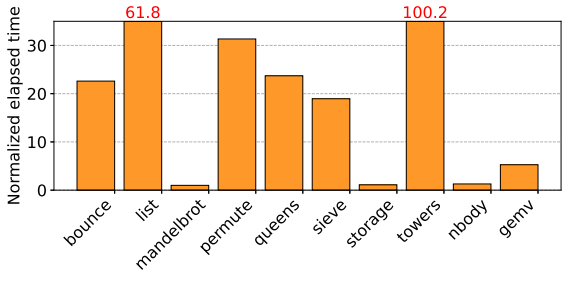


図 11 実行時間（C 言語プログラムに対する比）
Fig. 11 Elapsed time (normalized to C program)

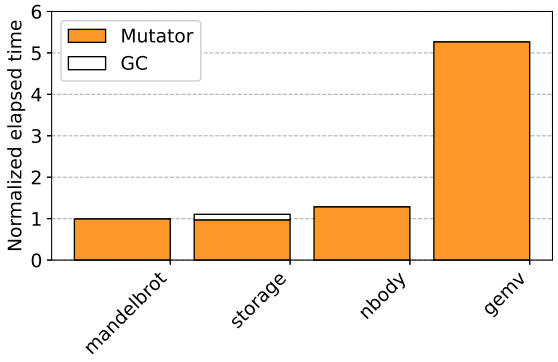


図 12 いくつかのプログラムに対する実行時間の詳細（C 言語プログラムに対する比）
Fig. 12 Close-up of elapsed time for selected programs (normalized to C program)

sieve はメソッド呼出しはないが，MRAM 中のオブジェクトに頻繁にアクセスする．素数の判定を行うための 5000 要素の配列は大きいので直接 MRAM に割り当てられる．そのため，MRAM 中のオブジェクトへのアクセスが頻繁になる．gemv も sieve と同様に大きな配列にアクセスするが，gemv は C 実装でも配列を MRAM に配置しているため，実行時間の比が小さい．

permute，queens，towers はメソッド呼出しも MRAM 中のオブジェクトへのアクセスも頻繁に行う．特に C 実装との比が大きい towers は，メソッド呼出しに伴い WRAM 中の Java スタックのオーバーフローとアンダーフローが 37.5×10^3 回/秒発生しており，そのたびに WRAM と MRAM の間で DMA 転送が発生している．

一方で，mandelbrot，storage，nbody は C 実装と実行時間があまり変わらなかった．mandelbrot はもともと計算インテンシブなプログラムであり，Java のプログラムも C 実装と似たような機械語にコンパイルされると考えられる．nbody もアクセサを手動でインライン展開したことにより，mandelbrot と同じような結果になった．

storage は MRAM のオブジェクトのアクセスやメソッド呼出しは他のプログラムと同程度に頻繁に行い，WRAM 中の Java スタックも頻繁にオーバーフローやアンダーフローをしている．しかし，実行時間は C 実装とあまり変わらない．これは，義経実装の方がメモリ割当てが高速であるためと考えている．storage は動的メモリ管理を必要とするプログラムである．C 実装ではスレッドセーフな buddy system アロケータを使いメモリの割当てと明示的な解放を行う．一方，義経ではメモリ割当ては TLAB に対するバンプポインタ割当てなので高速であり，解放には GC を使う．図 12 では実行時間の内訳として GC の時間を示している．GC の時間は全実行時間の 10.6%だった．これは全てマイナー GC で，メジャー GC は発生していなかった．

以上より，メソッド呼出しや MRAM にあるオブジェクトのアクセスは大きなオーバヘッドになっていることが分かる．これを緩和するためにメソッド呼出しを自動でインライン展開したり，頻繁にアクセスするオブジェクトを WRAM にキャッシュすることが考えられるが，これらは今後の課題とする．

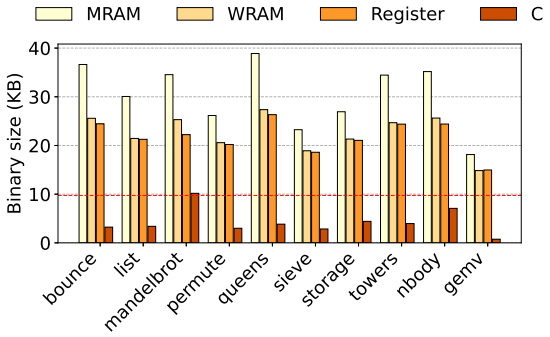


図 13 バイナリのサイズ
Fig. 13 Binary size.

4.2 バイナリサイズ

図 13 に義経実装と C 実装のバイナリのサイズを比較する．義経実装では，Java スタックの一部を WRAM に置いたり，Java のローカル変数やオペランドスタックをマシンレジスタに割り当てたことによりバイナリサイズが削減されたか確認するために，Java スタックを全て MRAM に置いたバージョン（MRAM），マシンレジスタには割り当てないがカレントフレームを WRAM に置いたバージョン（WRAM），マシンレジスタに割り当てるバージョン（Register）を作った．

義経実装では，ランタイムシステムを必要とする．ランタイムシステムの大きさを概算するために，空のメソッドを入口関数とするバイナリを作った．その結果，ランタイムシステムが 9.7 KB を占めていた．この 9.7 KB を図 13 に破線で示す．内訳はアロケータとマイナー GC 関連が 6.3KB，スケジューラ関数が 3.4KB だった．メソッドが空なのでオブジェクトのアクセスなどのランタイム関数はリンクされていないが，どのバイナリでも最低 9.7 KB はランタイムシステムに占められている．

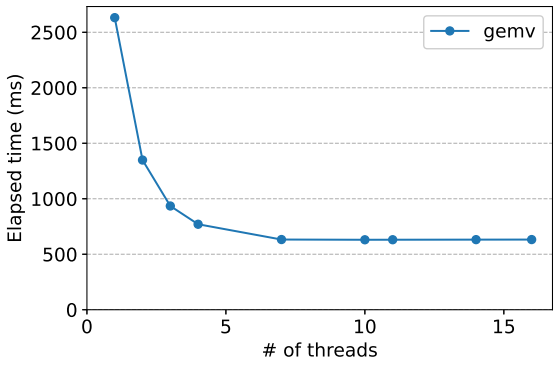


図 14 gemv のスケーラビリティ
Fig. 14 Scalability for gemv.

ランタイムシステムを差し引いた部分でも C 実装と比べると大きい．これは C 実装では 1 命令のメソッド呼出しやオブジェクトのアクセスが義経実装では長い命令列になるためと考えられる．それらが少ない mandelbrot や nbody では C 実装との差が他のプログラムより小さい．

Java のローカル変数やオペランドスタックの位置を変えたことによる効果については，カレントフレームを WRAM に配置することで，バイナリサイズを小さくできた．しかし，マシンレジスタに割り当てても mandelbrot を除いてバイナリサイズの削減効果は小さかった．マシンレジスタに割り当てると，メソッド呼出しの度にレジスタの内容を WRAM の Java スタックに退避する必要があり，それによりバイナリが大きくなる効果もあるためである．mandelbrot にはメソッド呼出しがほとんどないので，バイナリが小さくなった．

4.3 バイナリ入替えの時間

バイナリの入替えには，マイナー GC を含む WRAM の内容の退避，DPU の終了，新しいバイナリの転送，DPU の起動などのオーバヘッドがある．このオーバヘッドを計測するために，入口メソッドから空のメソッドを繰り返し呼び出すプログラムを作り，これらを別のバイナリにコンパイルして実行した．その結果，呼出しとリターンの 1 往復で 5.48 ms，1 回のバイナリ入替えは平均 2.74 ms かかった．これは 400 MHz で動作する DPU で約 1,000,000 サイクルに相当する．

この内訳を調べるために，実験に使った二つのバイナリをロードするだけの時間を計測した．その結果，1 回のロードにかかる時間がそれぞれ 2.72 ms と 2.55 ms であり，ほとんどの時間がバイナリのロードに使われていることが分かった．

4.4 スケーラビリティ

図 14 にスレッド数毎の gemv の実行時間を示す．4 スレッド程度まではほぼ線形にスケールしていることが分かる．しかし，その後は実行速度が向上しにくくなり，7 スレッドでほぼ飽和する．これは一つしかない DMA コントローラがボトルネックになっているためと考えられる．このベンチマークでは配列の 1 要素ごとに読み出す操作がほとんどなので，DMA の転送は 8 バイト単位のリードである．このとき DMA のスループットは 4 スレッド程度で飽和するという実測結果（2.1 節）と一致する．

5. 関連研究

Chen ら [5] は UPMEM PIM を分散システムとして利用するためのフレームワーク SimplePIM を提案した．SimplePIM は，分散システムフレームワークでよく用いられる `map`, `reduce`, `zip` などの操作を UPMEM PIM で実行するための C 言語の API を提供する．本研究は同様に分散システムとして利用するためのフレームワークだが，分散オブジェクトモデルに基づいている．オブジェクト指向言語である Java で，DPU にオブジェクトを配置し，それを透過的に操作するための API を提供する．

TornadoVM [22] は本研究と同様に Java プログラムの一部をアクセラレータにオフロードするためのフレームワークである．プログラマはアクセラレータで実行したいメソッドを指定する．TornadoVM はそのメソッドを OpenCL に変換し，続いてアクセラレータ向けのバイナリにコンパイルする．TornadoVM は OpenCL に変換できるような Java プログラムしか対象とせず，再帰呼出しやオブジェクトをサポートしない．本研究はこれらをサポートしている．

Zilli ら [23] はメモリが非常に限られた Java Card システムのために，Java プログラムのフットプリントを小さくする手法を提案した．この手法では Java プログラム中のバイトコード命令列をマクロ化したり，スーパー命令を導入することで冗長なバイトコード命令列を削減している．本研究ではバイトコードを一度 C 言語に変換してからコンパイルすることで，冗長な計算の削減を C コンパイラに任せている．

Java プログラムをバイナリに変換する研究として，Proebsting ら [18] による TOBA がある．TOBA は Java プログラムを C 言語のプログラムを経由してコンパイルするという点で本研究と同様である．一方で TOBA はアクセラレータではなく CPU で実行することを前提としているため，本研究のようにメモリの制約に対する対策は考えられていない．

プログラムの一部を入れ替えながら実行する手法は，命令キャッシュとしてスクラッチパッドメモリを備えるシステムにおいて，スクラッチパッドメモリのサイズを超えるプログラムを効率的に実行するための研究で利用されている [24], [25]．特に Baker ら [25] は，マルチコアのアクセラレータである Cell Broadband Engine（CBE）に対して実

際に実装して評価を行っているが，並列処理への言及はない．CBE の複数のコアの SPM は独立に書き換えられるため，特別な仕組みを用意する必要がないからと思われる．一方で DPU では複数のハードウェアスレッドが同じバイナリを実行するため，入替えの時に同期が必要になる．

トランポリン方式の関数呼出しは，繰返しを再帰呼出しで記述する傾向にある関数型言語においてスタックを伸ばさない手法の一つとして知られている [26], [27]．一般的には末尾呼出しでトランポリンを行うため，呼出し元にリターンするための継続は保存しない．一方，本研究では全てのメソッド呼出しをトランポリン方式で行うため，継続を Java スタックに保存する．

スタックを配置するメモリが小さいという問題に対して，メモリから溢れるときに関数フレームをより大きなメモリに移し，必要になったときに元のメモリに戻すという方法もある [28]．本研究ではトランポリン方式を使ってスタック溢れの問題とプログラムオーバーレイの両方に対応している．トランポリンでプログラムオーバーレイをすることは，CBE において Miller らも行っている [24]．

6. おわりに

本論文では，Java のプログラムの一部を UPMEM PIM にオフロードするフレームワーク「義経」のために，Java のメソッドをスクラッチパッドメモリを備え SPMD モデルの並列処理をサポートした PIM チップである DPU 向けにコンパイルする機構を開発した．

メソッドの呼出しをトランポリン方式にすることで実行スタックが伸びることを防ぎ，限られたスクラッチパッドメモリで深い再帰呼出しができるようにした．また，プログラムオーバーレイにより，プログラムメモリに収まらないプログラムも実行できるようにした．プログラムオーバーレイを伴う並列処理やバリア同期にも対応した．ヒープは新世代領域をスクラッチパッドメモリ，旧世代領域を DRAM に配置する 2 世代の構成にし，マイナー GC には GiY GC を用いた．

これらにより，プログラムによっては C 言語で同じ処理を記述したプログラムと同程度の速度で実行できた．しかし，メソッド呼出しや大きな配列のアクセス，古いオブジェクトへのアクセスが多いプログラムでは最大で約 100 倍遅くなった．一方で，これらのオーバヘッドの要因があるプログラムでも頻繁にメモリ割当てを行うプログラムは C 言語と同程度の速度で実行でき，本機構のメモリ割当ては高速であることが分かった．バイナリの入替えが起こると，約 1,000,000 サイクルの時間がかかることが分かった．バイナリサイズについては，約 10 KB のランタイムシステムに加え，コンパイルしたメソッド自身も C 言語より大きく，平均で 6.02 倍になった．

小さなメソッドの呼出しをインライン展開することで，実行時間もバイナリサイズも削減できると考えられる．また，頻繁にアクセスするオブジェクトをスクラッチパッドメモリにキャッシュするのも，高速化に寄与すると考えられる．これらは今後の課題である．

謝辞 本研究の実験に UPMEM PIM の搭載された計算機を利用させていただいた東京科学大学の藤木大地先生，ならびに，本研究グループで有益な助言をいただいた東京大学の塩谷亮太先生，立命館大学の穐山空道先生，電気通信大学の佐藤重幸先生に感謝申し上げます．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] Mutlu, O., Ghose, S., Gómez-Luna, J., Ausavarungnirun, R., Sadrosadati, M. and Oliveira, G. F.: A Modern Primer on Processing in Memory (2025).

[2] Falevoz, Y. and Legriel, J.: Energy Efficiency Impact of Processing in Memory: A Comprehensive Review of Workloads on the UPMEM Architecture, *Euro-Par 2023: Parallel Processing Workshops*, Cham, Springer Nature Switzerland, pp. 155–166 (2024).

[3] Gómez-Luna, J., Hajj, I. E., Fernandez, I., Giannoula, C., Oliveira, G. and Mutlu, O.: Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture, Technical report, Cornell University Library, arXiv.org (2021).

[4] Bernhardt, A., Koch, A. and Petrov, I.: Pimdb: From main-memory dbms to processing-in-memory dbms-engines on intelligent memories, *Proceedings of the 19th International Workshop on Data Management on New Hardware*, pp. 44–52 (2023).

[5] Chen, L.-C., Ho, C.-C. and Chang, Y.-H.: UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification, *2023 60th ACM/IEEE Design Automation Conference (DAC)*, IEEE, pp. 1–6 (2023).

[6] Lee, D., Hyun, B., Kwon, Y. and Rhu, M.: PIM-malloc: A Fast and Scalable Dynamic Memory Allocator for Processing-In-Memory (PIM) Architectures (to appear in HPCA 2026) (2025).

[7] de Dinechin, B. D., de Massas, P. G., Lager, G., Léger, C., Orgogozo, B., Reybert, J. and Strudel, T.: A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor, *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, Procedia Computer Science, Vol. 18, Elsevier, pp. 1654–1663 (online), DOI: 10.1016/J.PROCS.2013.05.333 (2013).

[8] Hatta, N., Tsunoda, S., Uchida, K., Ishitani, T., Koizumi, T., Shioya, R. and Ishii, K.: PEZY-SC4s : The Fourth Generation MIMD Many-core Processor with High Energy Efficiency and Flexibility for HPC and AI Applications, *2025 IEEE Hot Chips 37 Symposium (HCS)*, pp. 1–42 (online), DOI: 10.1109/HCS66204.2025.11154388 (2025).

[9] 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).

[10] Ichinose, K., Sato, S. and Ugawa, T.: Towards a Java Virtual Machine for Processing-In-Memory, *Compan-*

ion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025), Open Access Series in Informatics (OASIcs), Vol. 134, Dagstuhl, Germany, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 2:1–2:5 (online), DOI: 10.4230/OASIcs.Programming.2025.2 (2025).

[11] Morimoto, R., Ichinose, K. and Ugawa, T.: Gray-in-Young: A Generational Garbage Collection for Processing-in-Memory, *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management*, ISMM '25, New York, NY, USA, Association for Computing Machinery, p. 122–133l (online), DOI: 10.1145/3735950.3735961 (2025).

[12] Ichinose, K. and Ugawa, T.: プログラムメモリの小さな Processing-in-Memory 向け Java to C コンパイラ (in Japanese), *Proceedings of the 41st JSSST Annual Conference* (2024).

[13] Ichinose, K. and Ugawa, T.: 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 (in Japanese), *Proceedings of the 41st JSSST Annual Conference* (2024).

[14] Ben-David, N. and Blelloch, G. E.: Fast and fair randomized wait-free locks, *Distributed Comput.*, Vol. 38, No. 1, pp. 51–72 (online), DOI: 10.1007/S00446-024-00474-4 (2025).

[15] Lebsack, C. S. and Chang, J. M.: Using Scratchpad to Exploit Object Locality in Java, *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, USA, IEEE Computer Society, p. 381–386 (online), DOI: 10.1109/ICCD.2005.111 (2005).

[16] Chong, K. F., Ho, C. Y. and Fong, A. S.: Pretenuring in Java by Object Lifetime and Reference Density Using Scratch-Pad Memory, *Proceedings of 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing*, pp. 205–212 (online), DOI: 10.1109/PDP.2007.67 (2007).

[17] Sekiguchi, T., Masuhara, H. and Yonezawa, A.: A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation, *Proceedings of the Third International Conference on Coordination Languages and Models*, COORDINATION '99, Berlin, Heidelberg, Springer-Verlag, p. 211–226 (1999).

[18] Proebsting, T. A., Townsend, G. M., Bridges, P. G., Hartman, J. H., Newsham, T. and Watterson, S. A.: Toba: Java for Applications-A Way Ahead of Time (WAT) Compiler., *COOTS*, Vol. 97, pp. 41–54 (1997).

[19] Cheney, C. J.: A nonrecursive list compacting algorithm, *Commun. ACM*, Vol. 13, No. 11, p. 677–678 (online), DOI: 10.1145/362790.362798 (1970).

[20] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F. M.: On-the-fly garbage collection: an exercise in cooperation, *Commun. ACM*, Vol. 21, No. 11, p. 966–975 (online), DOI: 10.1145/359642.359655 (1978).

[21] Marr, S., Daloze, B. and Mössenböck, H.: Cross-Language Compiler Benchmarking—Are We Fast Yet?, *Proceedings of the 12th Symposium on Dynamic Languages*, DLS'16, ACM, pp. 120–131 (online), DOI: 10.1145/2989225.2989232 (2016).

[22] Fumero, J., Papadimitriou, M., Zakkak, F. S., Xekalaki, M., Clarkson, J. and Kotselidis, C.: Dynamic application reconfiguration on heterogeneous hardware, *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 165–178 (2019).

[23] Zilli, M., Raschke, W., Weiss, R., Steger, C. and Loinig, J.: A light-weight compression method for Java Card technology, *ACM SIGBED Review*, Vol. 11, No. 4, pp. 13–18 (2015).

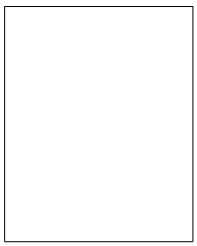
[24] Miller, J. E. and Agarwal, A.: Software-based instruction caching for embedded processors, *ACM SIGARCH Computer Architecture News*, Vol. 34, No. 5, pp. 293–302 (2006).

[25] Baker, M. A., Panda, A., Ghadge, N., Kadne, A. and Chatha, K. S.: A performance model and code overlay generator for scratchpad enhanced embedded processors, *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 287–296 (2010).

[26] Schinz, M. and Odersky, M.: Tail call elimination on the Java Virtual Machine, *Electronic Notes in Theoretical Computer Science*, Vol. 59, No. 1, pp. 158–171 (2001).

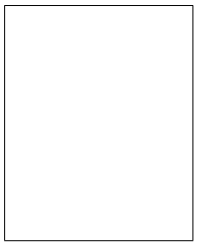
[27] Óli Bjarnarson, R.: Stackless scala with free monads, *Scala Days* (2012).

[28] Bai, K., Shrivastava, A. and Kudchadker, S.: Stack data management for limited local memory (LLM) multi-core processors, *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, IEEE, pp. 231–234 (2011).



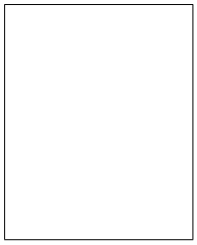
一野瀬 知輝

2001 年生．2023 年東京大学工学部電気電子工学科卒業．2025 年東京大学情報理工学系研究科創造情報学専攻修士課程修了．



森本 龍

2000 年生．2023 年東京工業大学工学院システム制御系卒業．2026 年東京大学情報理工学系研究科創造情報学専攻修士課程修了．



鵜川 始陽 （正会員）

1978 年生．2000 年京都大学工学部情報学科卒業．2002 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．2005 年同専攻博士後期課程修了．同年京都大学大学院情報学研究科特任助手．2008 年電気通信大学助教．2014 年高知工科大学准教授．2020 年より東京大学大学院情報理工学系研究科准教授．博士 (情報学)．プログラミング言語とその処理系に関する研究に従事．情報処理学会 2012 年度山下記念研究賞受賞．