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

一野瀬 知輝　鵜川 始陽

Processing-in-Memory(PIM) とは DRAM チップ上にプロセッサ（以下メモリプロセッサ）が配置されたアーキテクチャであり，そこに計算の一部をオフロードすることで CPU と DRAM との間のデータ移動による遅延や消費電力を低減できることが期待されている．PIM アプリケーションの開発効率向上のため，我々は 1 つの Java のプログラムで CPU とメモリプロセッサの計算を記述できるシステムを開発している．本研究ではオフロードするメソッドを事前にメモリプロセッサの機械語にコンパイルするシステムを提案する．メモリプロセッサのプログラムメモリは小さいため，複数のバイナリにメソッド単位で分割してコンパイルし，それらをプログラムオーバーレイにより実行する．このとき，呼出し関係にあるメソッドは同じバイナリにコンパイルする．提案システムをインタプリタを用いた先行研究と比較した．先行研究で用いられていたベンチマークではバイナリの入れ替えが起きず，実行は 31% 高速だった．また，プログラムオーバーレイのオーバーヘッドの内訳を調べ，バイナリ入れ替えの割合が大きいことがわかった．

1　はじめに

Processing-in-Memory (PIM) は DRAM チップ上にプロセッサ（以下メモリプロセッサ）を配置したアーキテクチャである．DRAM 内のデータベース検索などのデータインテンシブな計算をメモリプロセッサにオフロードすることで，CPU と DRAM の間のデータ転送に起因する遅延や消費電力のボトルネックが低減できることが報告されている [2]．PIM アプリケーションの開発効率を向上させるため，我々は Java 言語の 1 つのプログラム内に CPU とメモリプロセッサの計算を記述できるフレームワークを開発している．

先行研究 [5] で開発したプロトタイプシステムでは，オフロードする Java プログラムのバイトコードを，機能を制限した Java インタプリタをメモリプロセッサで動作させて逐次実行していた．

しかしこの方式には，次の 2 つの問題がある．

- インタプリタにはバイトコードのフェッチやディスパッチなどのオーバーヘッドがある．
- 完全なインタプリタはメモリプロセッサのプログラムメモリに収まらない．

そこで本研究では，オフロードする Java プログラムを事前にメモリプロセッサの機械語にコンパイルするシステムを提案する．コンパイル後のプログラムのサイズは，もとの Java プログラムの規模によって変化する．プログラムメモリに収まらない大規模なプログラムの実行も可能にするために，メソッド単位でプログラムを分割してから，静的コールグラフをもとに呼出し関係にあるメソッド同士を同じバイナリにまとめる．こうしてできた複数のバイナリをプログラムオーバーレイにより必要に応じてバイナリを入れ替えながら実行する．

提案システムを評価するために，我々は 2 つの実験を行った．先行研究 [5] と同じベンチマークを用いて提案のコンパイル方式による実行時間を比較した結果，31%の短縮が見られた．また，プログラムオーバーレイのオーバーヘッドには，バイナリ入れ替えそ

---

* Java to C Compiler for Processing-in-Memory with Small Program Memory.

Kazuki Ichinose, Tomoharu Ugawa, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

のもののオーバーヘッドと，入れ替えが起きても実行の継続を可能にする仕組みにより普段の実行で発生するオーバーヘッドがある．マイクロベンチマークを用いてこれらの割合を調べた結果，バイナリ入れ替えそのもののオーバーヘッドが支配的であることがわかった．

2 問題

2.1 UPMEM システム

UPMEM システム [8] は，PIM デバイスを搭載した，現在一般に利用可能なシステムの 1 つである．UPMEM システムは CPU，従来の DRAM，およびメモリプロセッサを搭載した DRAM である UPMEM PIM モジュールによって構成される．UPMEM PIM モジュールの各 DRAM チップを以下では PIM チップと呼ぶ．

PIM チップの詳細を図 1 に示す．PIM チップにはメモリプロセッサとして DRAM Processing Unit(DPU) が積層されている．各 DPU は専用のプログラムメモリ (IRAM)，スクラッチパッドメモリ (WRAM)，メインメモリ (MRAM) を備えていて，いずれのメモリも CPU とデータのやりとりができる．

各メモリは異なる特徴を持つ．IRAM には CPU からプログラムを書き込み，実行することができるが，24KB と小さいためプログラムの大きさに制限がある．WRAM は 64KB の高速なメモリである．MRAM は 64MB と容量が大きいが，DPU は DMA コントローラを通してアクセスする必要があり，アクセスのオーバーヘッドが大きい．それぞれのメモリの特性を考慮しながらプログラミングをする必要がある．

2.2 PIM プログラミングフレームワーク

我々は，CPU と DPU で実行されるプログラムの両方を Java で 1 つのプログラム内に記述することができるフレームワークを開発している．このフレームワークは CPU を中心とした分散オブジェクトモデルに基づいていて，CPU から PIM チップ内にオブジェクトを作成し，そのオブジェクトのメソッドを DPU で実行することができる．DPU から CPU 上のオブジェクトに対するメソッド呼出しをすることはできない．

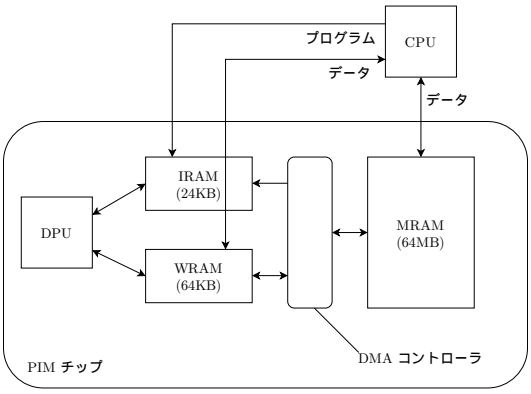

図 1　PIM チップ構成

我々のフレームワークは CPU から指定の DPU の MRAM に指定のクラスのインスタンスを作成する API(`createObject`)，MRAM 内のオブジェクトのメソッドを DPU で実行する API(`invokeMethod`)，`invokeMethod` で実行したメソッドの戻り値を取得する API(`getIntReturnValue` など) を提供する．`createObject` は，作成したインスタンスに対応するハンドルを返す．`invokeMethod` はこのハンドルを介してメソッドを呼出す．

我々のフレームワークを利用した PIM アプリケーションの例を図 2 に示す．このプログラムは，図 3 に示す二分探索木クラスで PIM チップ中に木を構築し，その木を探索する．まず，`createObject` により MRAM 内にキーを 123，値を 456 としてノードを作成する．そのノードのハンドルに対して，`invokeMethod` により `insert` メソッドを繰り返し呼び出すことで，MRAM 内に木を構築する．続いて `search` メソッドを呼出して，キーが 100 のノードを DPU で探索する．最後に `getIntReturnValue` により探索結果を取得すると，はじめに挿入したノードの値である 200 が返される．

先行研究 [5] ではさらに，ハンドルをラップするプロキシオブジェクトを作ることで，透過的に DPU にあるオブジェクトのメソッドを呼出せるようにしている．これは本研究の範囲を超えるので説明は省略する．

```
Handle handle = UPMEM.createObject("TreeNode",
  123, 456);

UPMEM.invokeMethod(handle, "insert",
  100, 200);
...

UPMEM.invokeMethod(handle, "search", 100);
int result = UPMEM.getIntReturnValue(); // 200
```

図 2　PIM チップの中に木を構築して探索するプログラムの例

```
public class TreeNode {

  public void insert(int k, int v) {
    / * Insert a new node with key=k, val=v. */
  }

  public int search(int k) {
    if (k == getKey())
      return getVal();
    if (k < getKey() && getLeft() != null) {
      return getLeft().search(k);
    } else if (k >= getKey() &&
        getRight() != null) {
      return getRight().search(k);
    }
    return -1;
  }
}
```

図 3　PIM チップの中で扱う二分探索木クラス

2.3　PIM アプリケーションのインタプリタ実行

`invokeMethod` の際には，DPU で Java プログラムを実行する仕組みが必要である．先行研究のシステムでは PIM チップ中で扱うクラスのバイトコードを MRAM にロードし，機能を制限した Java インタプリタを DPU で動作させてバイトコードを逐次解釈，実行していた．

このシステムでは，インタプリタの実行時オーバーヘッドが大きいという問題がある．具体的には，バイトコード命令の実行のたびに DMA コントローラを通して MRAM から命令をフェッチするコストがある．また，命令をディスパッチするコストもある．

これに加えて，完全なインタプリタは IRAM に収まらないという問題もある．先行研究では二分探索木クラスを扱うために最低限必要になるバイトコードであるメソッド呼出しや整数・オブジェクトに対する命令のみを実装しているが，プログラムサイズは 9KB に達する．

3　提案システム

本研究ではオフロードするメソッドを事前に DPU の機械語にコンパイルするシステムを提案する．メソッドの各バイトコード命令を，それと等価な機械語に置き換えたバイナリを生成することで，命令のフェッチとディスパッチのオーバーヘッドを削減する．また，バイナリが IRAM に収まらないサイズになっても実行できるように，プログラムをメソッド単位で分割してコンパイルし，プログラムオーバーレイによりバイナリを入れ替えながら実行する．

3.1　DPU の機械語へのコンパイル方法

我々のシステムでは Java プログラムのバイトコードを，一度 C 言語のプログラムにコンパイルし，得られた C 言語のプログラムをコンパイルすることで DPU 用のバイナリを生成する．バイトコードから C 言語へのコンパイルでは各バイトコード命令をその命令と等価な C 言語のプログラム片に置き換える．C 言語からバイナリへのコンパイルは UPMEM のソフトウェア開発環境 †1 が提供する DPU 用の C コンパイラを使用する．

1 つの Java のメソッドは 1 つの C 言語の関数にコンパイルする．関数の中には，各バイトコード命令と等価な C 言語のプログラム片が並ぶ．バイトコードから C 言語へのコンパイルを，図 4 に示す `search` メソッドのバイトコードを例に説明する．`iload_1` と `aload_0` はそれぞれ 1 番目のローカル変数を整数として，0 番目のローカル変数を参照として Java スタックにプッシュする命令である．また `invokevirtual` は仮想メソッドを解決して呼び出す命令である．これをコンパイルした C 言語プログラム片を図 5 に示す．`iload_1` はローカル変数の 1 番目を読み出し，Java スタックの一番上に格納してから スタックポインタ (`SP`) をインクリメントする命令にコンパイルする．ここで，local は Java スタック中のローカル変

†1　https://sdk.upmem.com/

```
public int search(int);
  0: iload_1
  1: aload_0
  2: invokevirtual "getKey"
  ...
```

図 4　search メソッドのバイトコード

```
  /* iload_1 */
  stack[SP++] = local[1];

  /* aload_0 */
  stack[SP++] = local[0];

  /* invokevirtual "getKey" */
  stack[SP++] = BP;
  stack[SP++] = 0; // Return label
  stack[SP++] = ID_search;

  OBJ_ADDR reveiver = stack[SP - 3 -
    CALLEE_ARGUMENT_SIZE];
  trampoline_info->method_id =
    resolve_virtual(receiver, INDEX_getKey);
  trampoline_info->is_call = true;
  return;

RETURN_LABEL0:
  ...
```

図 5　バイトコードのコンパイル例

数の領域を指すポインタである．aload_0 も同様にコンパイルする．続く invokevirtual をコンパイルした部分は 3.2 節で説明する．

3.2　トランポリンによるプログラムオーバーレイ

メソッドを呼び出す invokevirtual 命令はプログラムオーバーレイを伴う場合がある．つまり，実行中のバイナリに呼出し先のメソッドが存在しなければ，呼出し先のメソッドが存在するバイナリに入れ替えてから呼び出す．PIM チップではバイナリを入れ替える際に WRAM の内容は初期化されるため，WRAM に配置される C 言語の実行スタックが消去されても問題ないように対策が必要である．

そこで，我々のシステムでは実行に必要な情報は全て MRAM に保存する．まず Java のスタックは MRAM に配置する．さらに， Java のメソッド呼出しは，いわゆる「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，最初の Java のメソッド実行前に，特別な「トランポリン関数」を実行し，そこから Java のメソッドをコンパイルした関数を呼び出す．そのうえで， Java のメソッド呼出しもリターンも，トランポリン関数にリターンする C 言語のプログラム片にコンパイルする．トランポリン関数は呼出し先やリターン先のメソッドに対応する C 言語の関数を呼出す．これにより，WRAM 中の実行スタックは伸びず，保存する必要がなくなる．

図 6 に示す例をもとに具体的に説明する．この図は， search メソッドが再帰的に呼ばれ，その後バイナリ入れ替えを伴って getKey メソッドが呼ばれたときの C 言語の実行スタックと Java のスタックの様子を示している．まず，search メソッドが search メソッドを呼出すと，バイナリの入れ替えは発生せず，トランポリン関数にリターンしてから再度 search の関数を呼出す．次に getKey メソッドが呼出される．getKey の関数は実行中のバイナリに入っていなかったとする．このときは CPU に制御を戻して getKey の関数が入ったバイナリに入れ替え，それを実行する．このとき，Java のスタックは MRAM にあるため入れ替えの前後で維持される．また，CPU へ制御を戻すのはトランポリン関数からリターンすることで行うため，WRAM 中の実行スタックには何も残らない．getKey メソッドから search メソッドにリターンするときは再びバイナリを入れ替えて実行する．

3.2.1　メソッド呼出し

invokevirtual は以下の情報を Java スタックに積んで，仮想メソッドの解決をした後，トランポリン関数にリターンする．ここで積む情報は，invokevirtual で呼出されたメソッドからリターンした後，現在のメソッドの実行を継続するために必要になる．

- 現在のベースポインタ (BP)
- リターン後の再開位置を表す ID
- 現在のメソッド ID

リターン後に実行を再開する位置，つまりメソッド呼出しの直後にはラベルを付けておき，リターン後にその位置にジャンプできるようにする．ラベルはそのまま Java スタックに積めないので一意に ID をふり，それを積む．図 5 では現在の BP，リターン後の実行再開位置のラベル RETURN_LABEL0 の ID である 0，現在実行している search メソッドの ID を Java

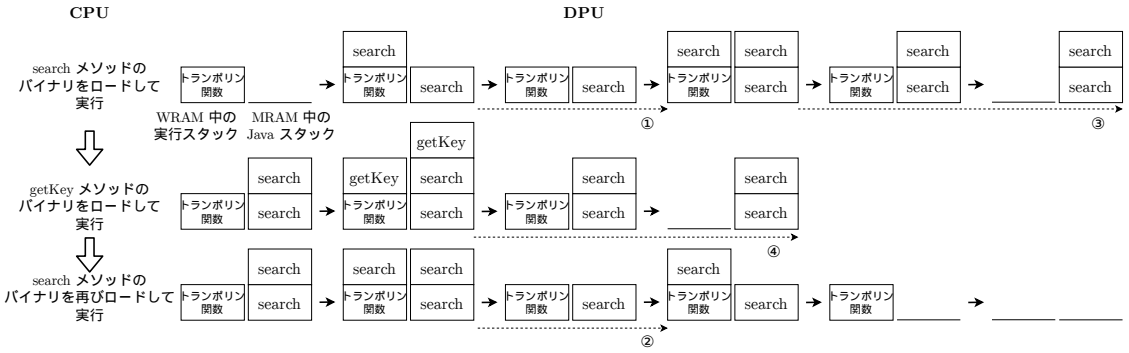

図 6　プログラムオーバーレイ時のスタックの様子

スタックに積んでいる．

トランポリン関数にリターンするときには，以下に示す情報を `trampoline_info` という構造体に書き込む．

- 次に実行すべきメソッドの ID (`method_id`)
- その実行がメソッド呼出しかリターンか (`is_call`)
- 実行再開位置の ID (`label_id`)

`method_id` には仮想メソッドを解決して得られたメソッド ID を書き込む．仮想メソッドを解決する方法は 3.2.5 節で詳しく説明する．図 5 では `getKey` メソッドの解決をして得られた ID を `method_id` 書き込んでいる．`is_call` には呼出しであることを示す `true` を書き込む．`label_id` はメソッド呼出しでは書き込まない．

トランポリン関数は `trampoline_info` に書かれた次に実行すべきメソッドが同じバイナリにコンパイルされていれば，それを呼出す．図 6 の ① の矢印がそれを示している．

3.2.2　リターン

リターンの場合は，Java スタックを巻き戻した後，`invokevirtual` で Java スタックに積んだ情報を取り出して，`trampoline_info` に書き込み，戻り値を Java スタックに積んでトランポリン関数にリターンする．トランポリン関数は，`trampoline_info` に書かれた情報をもとにリターン先メソッドをコンパイルした関数を呼出す．図 6 の② の矢印がその様子を示している．

```
1  /* ireturn */
2  int return_value = stack[SP - 1];
3  SP = BP;
4  BP = stack[SP + ARGUMENT_SIZE];
5  trampoline_info->label_id =
6    stack[SP + ARGUMENT_SIZE + 1];
7  trampoline_info->method_id =
8    stack[SP + ARGUMENT_SIZE + 2];
9  trampoline_info->is_call = false;
10 stack[SP++] = return_value;
11 return;
```

図 7　`ireturn` 命令をコンパイルした C 言語のプログラム片

具体例として図 5 で呼出された `getKey` メソッドからリターンする場合を説明する．`getKey` メソッドは `int` 型を返す `ireturn` 命令によりリターンする．コンパイルしたプログラム片を図 7 に示す．まず，Java スタックの一番上に積まれている返り値を取り出す．その後，Java スタックを巻き戻すために `SP` と `BP` を設定し，呼出し元の `BP`，実行再開位置の ID，そしてメソッド ID を Java スタックから取り出し `trampoline_info` と `BP` を設定する．また，リターンであることを示すために `is_call` に `false` を設定する．最後に返り値を Java スタックに積んでトランポリン関数にリターンする．

3.2.3　関数プロローグ

Java のメソッドをコンパイルした関数の冒頭には図 8 に示すプロローグを挿入し，メソッド呼出しとリターンのそれぞれに応じた処理を行う．まず `trampoline_info` の `is_call` をもとにメソッド呼

```
1  if (trampoline_info->is_call) {
2    BP = SP - 3 - ARGUMENT_SIZE;
3    SP = BP + FRAME_SIZE;
4  } else { /* Return */
5    switch (trampoline_info->return_label) {
6    case 0: goto RETURN_LABEL0;
7    case 1: goto RETURN_LABEL1;
8    ...
9  } }
```

図 8　メソッドをコンパイルした関数のプロローグ

出しかリターンかを判定する．呼出しの場合は Java のスタックフレームを設定する．`invokevirtual` で引数とリターンのための 3 つの情報を Java スタックに積んでいるので，`BP` は呼出し元の `SP` よりその分だけ戻した位置に設定する．`SP` は `BP` にフレームの大きさを足した位置に設定する．引数の数とフレームの大きさはコンパイル時に決まる．その後は先頭のバイトコードの処理に移る．リターンの場合は，`trampoline_info` に保存された実行再開位置のラベルにジャンプしてバイトコードの処理を継続する．

3.2.4　バイナリ入れ替え

メソッド呼出しでバイナリ入れ替えが必要かはトランポリン関数が確認し，必要ならばバイナリを入れ替えてから関数を呼び出す．バイナリを入れ替えるときは CPU に次のメソッドの ID を知らせる．CPU はそれに基づきロードするバイナリを決める．メソッド ID とバイナリの対応もコンパイル時に生成する．図 6 では，③ がバイナリ入れ替えを伴うメソッド呼び出しで，④ がバイナリ入れ替えを伴うリターンである．

バイナリ入れ替えを実現するトランポリン関数を図 9 に示す．5 行目でトランポリン関数は次に実行されるメソッドの ID を読み出し，そのメソッドをコンパイルした関数が現在のバイナリに含まれるか確認する．含まれる場合はその関数を呼び出し，含まれなければコンパイルした関数が含まれるバイナリに入れ替える (16 行目)．含まれるかどうかの確認は，コンパイル時にバイナリごとに生成するテーブルで行う．

バイナリを入れ替える前に，トランポリン関数は `trampoline_info` を MRAM に保存しておく．保存した内容は，次のバイナリのトランポリン関数の 3

```
1  void trampoline() {
2    trampoline_info_t trampoline_info;
3    read_from_mram(&trampoline_info);
4    while (1) {
5      if (is_in_same_binary(
6          trampoline_info.method_id)) {
7        switch (trampoline_info.method_id) {
8        case ID_search:
9          search(&trampoline_info); break;
10       case ID_getKey:
11         getKey(&trampoline_info); break;
12       ...
13       }
14     } else {
15       write_to_mram(&trampoline_info);
16       change_binary(trampoline_info->method_id);
17 } } }
```

図 9　トランポリン関数

行目で復元される．また，CPU が `invokeMethod` でメソッドを呼出すときは `trampoline_info` を設定して MRAM に書き込んでおく．これも 3 行目で読み出す．

3.2.5　メソッドの ID と仮想メソッドの解決

メソッドには PIM アプリケーション全体で一意な ID をコンパイル時に割り当てる．クラスの継承関係があっても，子クラスのメソッドと親クラスのメソッドは別の ID を割り当てる．仮想メソッドの解決に用いる vtable の各メソッドのインデックスには対応するメソッドの ID が格納される．図 5 に示すとおり `invokevirtual` 命令の中で `resolve_virtual` は，レシーバの vtable の指定されたインデックスにあるメソッドの ID を返す．vtable はアプリケーションの開始時にフレームワークが MRAM に書き込む．

3.3　静的コールグラフに基づくプログラムの分割

我々のシステムでは呼出し関係にあるメソッドを同じバイナリにまとめる．これにより，プログラムオーバーレイの際のバイナリ入れ替えの回数を削減してオーバーヘッドを減らす．呼出し関係を調べるために，SootUp [6] を使って静的コールグラフを作成する．

静的コールグラフをたどり，見つけたメソッドを同じバイナリにまとめてコンパイルする．これにより，あるメソッドから直接呼び出されるメソッドだけでなく，間接的に呼び出されるメソッドも同じバイナリに

まとめることができる．こうすることでできるだけバイナリの入れ替え回数を減らす．

子クラスがある場合は，子クラスのメソッドも同じバイナリに含める．例えば `TreeNode` クラスの子クラスが `getKey` メソッドをオーバーライドしている場合，それも同じバイナリにコンパイルすることで，二分木の中に子クラスのノードがある場合でも同じバイナリを使うことができ，バイナリ入れ替えを削減できる．

4 実験

本章では提案システムの性能を評価するために行った 2 つの実験の結果を示す．1 つ目の実験では，コンパイル方式である提案システムとインタプリタ方式である先行研究のシステムの実行時間を比較した．評価には先行研究で用いられたベンチマークを使用した．2 つ目の実験では，プログラムオーバーレイによるオーバーヘッドと，その内訳を調べた．

実験は以下の構成の UPMEM システムで行った．

- CPU: Intel Xeon Silver 4215 (2.50GHz)
- DRAM: DDR4-2666 256GB
- UPMEM PIM モジュール (バージョン v1A) ×20
  - DPU (350 MHz) 24 スレッド ×2560
- UPMEM SDK: 2023.2.0
- Java: OpenJDK 17.0.11

DPU は 1 つのみ使用した．また，DPU は 24 スレッドを持つが，1 スレッドのみを使用した．複数の DPU の使用とマルチスレッド化は今後の課題である．

4.1 インタプリタとの実行時間の比較

コンパイルによってインタプリタよりどの程度高速になったかを評価するために，提案システムと先行研究のインタプリタを用いたシステムでの実行時間を比較した．

先行研究で用いられていたベンチマークは二分探索木の探索である．木の根から一定の深さまでのノードは従来の DRAM に配置され，メソッド呼出しは CPU で処理される．一定の深さよりも深いノードは MRAM に配置され，メソッド呼出しは DPU で処理される．ランダムに生成したキーと値を持つノードを木に追加し，検索を行う．検索を繰り返し行い，その間の実行時間を計測する．

この実験では，合計 1000 ノードからなる二分探索木を作成し，深さ 5 までのノードを CPU で処理し，それより深いノードを DPU で処理した．検索は 500 回行い，その実行時間を計測した．このベンチマークでは，同じメソッドを繰り返し実行するためバイナリの入れ替えは発生しなかった．

結果を表 1 に示す．実行時間は 31% 短縮された．この原因を調べるために，探索の間の DPU での命令数とサイクル数を計測した．この結果も表 1 に示す．提案システムでは，命令数とサイクル数が先行研究のシステムと比べて大幅に減っている．命令フェッチやディスパッチのために実行していた命令がコンパイルすることによって不要になったためと考えられる．1 命令あたりにかかる平均サイクル数にはほとんど差がないことから，命令数の減少には MRAM アクセス命令も含まれると考えられる．平均サイクル数は主に MRAM アクセスの際の DMA によって大きくなるからである．

4.2 プログラムオーバーレイによるオーバーヘッド

プログラムオーバーレイを可能にするためにかかるオーバーヘッドは，次の 3 つに細分化できる．

- CPU に制御を戻しバイナリを入れ替えるオーバーヘッド
- トランポリンによるオーバーヘッド
- Java スタックを MRAM に置くことによるオーバーヘッド

これらのオーバーヘッドを調べるために，それぞれのオーバーヘッドを分離して実験を行った．ベンチマークには，図 10 に示す Tarai クラスを用いた．`exec` メソッドは，Tarai 関数 [7] として知られる，再帰的な関数呼出しを繰り返す関数である．`createObject` により MRAM 内に Tarai クラスのインスタンスを生成し，`exec` メソッドを呼出した．CPU から呼出したときの `exec` の引数は `(8, 4, 1)` であり，`exec` メソッドは 2345 回呼び出された．

ベンチマークプログラムをコンパイルして得た C 言語コードに対して，3 種類変更を加えた．それぞ

表 1　提案システムと先行研究のシステムにおける二分探索木の平均実行時間および，DPU の平均命令数と平均サイクル数

システム	平均実行時間 (µs)	平均命令数	平均サイクル数	命令あたりサイクル数
提案	53.7	2.19×10^3	4.12×10^4	18.8
先行研究	70.3	5.41×10^3	1.02×10^5	18.9

```
1 public class Tarai {
2   public int exec(int x, int y, int z) {
3     if (x <= y)
4       return y;
5     else
6       return exec(exec(x - 1, y, z),
7         exec(y - 1, z, x), exec(z - 1, x, y));
8 } }
```

図 10　ベンチマークに使用した Tarai クラス

れ，上記で細分化したオーバーヘッドを調べるための変更である．その上で，変更を加えていないプログラムを含む次の 4 つのプログラムを実行し，その実行時間を計測した．

- 変更を加えていないプログラム (proposed)
- `is_in_same_binary` の結果によらず CPU に制御を戻し，バイナリを IRAM にロードし直すプログラム (always_change)
- トランポリンを利用せず，直接関数を呼び出すプログラム (no_trampoline)
- Java スタックを WRAM に置くプログラム (stack_in_wram)

結果を図 11 に示す．always_change はオーバーヘッドが大きすぎるため，グラフに含めていない．always_change は proposed に比べて 210 倍時間がかかった．proposed と no_trampoline の実行時間に大きな差はなく，no_trampoline は stack_in_wram より 2 倍時間がかかった．これらの結果から，バイナリ入れ替えに備えて MRAM に Java スタックを置くオーバーヘッドは大きいが，トランポリンによるオーバーヘッドは比較的小さいことがわかる．また，バイナリ入れ替えが発生した場合のオーバーヘッドは極めて大きく，バイナリ入れ替えを避けることが重要であることがわかる．

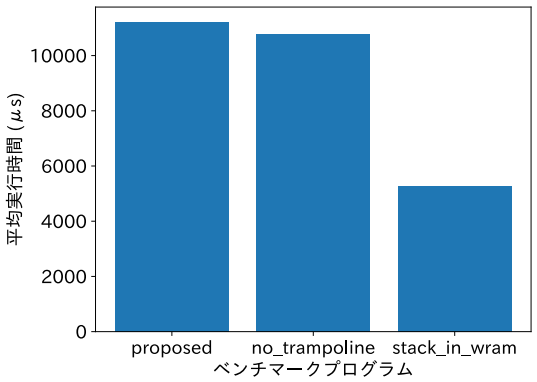

図 11　オーバーヘッドが異なる複数の Tarai ベンチマークプログラムに対する平均実行時間

5　関連研究

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研究として TornadoVM [4] がある．TornadoVM は GPU や FPGA などの異種アクセラレータにオフロードする計算を Java で記述するためのフレームワークである．プログラマは int などのプリミティブ型に対する計算をメソッドとして記述でき，TornadoVM はそのメソッドを OpenCL のコードにコンパイルして実行する．本研究は，TornadoVM が対象としていない PIM チップを対象としている点，計算がプリミティブ型に限られず，PIM チップ内のオブジェクトを扱える点で異なる．

PIM のプログラミングモデルに関して，Chen [3] らは PIM と 分散コンピューティングとの類似性に着目し，gather などの通信機能や map などのイテレータ処理を個々の PIM チップの存在を意識することなく利用できる C 言語フレームワークを提案している．このフレームワークは PIM のプログラムの記述を容易にする一方，オブジェクト指向ではない点，イテレータの処理ごとに別々のプログラムを記述する

必要がある点で本研究とは異なる．

プログラムメモリが小さいプロセッサに対してプログラムオーバーレイを適用する研究は，近年では CELL Broadband Engine (CBE) アーキテクチャに対して行われてきた．Baker [1] らは分割したプログラムをアドレスの重複を許してメモリにマップする際のアルゴリズムを提案している．アドレスが重複した部分を必要なときに入れ替えながら実行する状況を想定し，入れ替えのコストのモデルに基づきオーバーヘッドが少なくなるようなマッピングを生成する．本研究では静的コールグラフに基づき呼出し関係にあるメソッドを単にまとめてプログラムメモリにマップしたが，Baker らのアルゴリズムによりさらなる性能の向上の可能性がある．ただし対象とするアーキテクチャの違いからそのまま適用することは難しい．これは今後の課題とする．

6 まとめ

本論文では，Java プログラムの一部を DPU にオフロードする上で課題となっていたインタプリタの実行時オーバーヘッドとプログラムサイズの問題に対して，事前に DPU の機械語にコンパイルして実行するシステムを提案した．静的コールグラフ解析により呼出し関係にあるメソッドをまとめた複数のバイナリにコンパイルし，それらのバイナリをプログラムオーバーレイにより入れ替えながら実行することで，プログラムサイズが小さい PIM チップでも実行できるようにしている．

評価では，提案システムが先行研究のインタプリタを用いたシステムに対して 31% の実行時間の短縮を達成した．また，プログラムオーバーレイによるオーバーヘッドの内訳を明らかにし，バイナリ入れ替えによるオーバーヘッドが支配的で，その次に MRAM 中の Java スタックにアクセスするオーバーヘッドが大きいことを示した．

今後の課題として，Baker [1] らのアルゴリズムによりバイナリ入れ替えによるオーバーヘッドを抑えることや，WRAM に Java スタックの一部をキャッシュして実行時の MRAM アクセスをできるだけ減らすことが挙げられる．

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

参 考 文 献

[1] 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*, 2010, pp. 287–296.

[2] 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*, 2023, pp. 44–52.

[3] Chen, J., Gómez-Luna, J., Hajj, I. E., Guo, Y., and Mutlu, O.: Simplepim: A software framework for productive and efficient processing-in-memory, *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, 2023, pp. 99–111.

[4] 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*, 2019, pp. 165–178.

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

[6] Karakaya, K., Schott, S., Klauke, J., Bodden, E., Schmidt, M., Luo, L., and He, D.: SootUp: A Redesign of the Soot Static Analysis Framework, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2024, pp. 229–247.

[7] Takeuchi, I.: On a recursive function that does almost recursion only, *Memorandum, Musahino Electrical Communication Laboratory, Nippon Telephone and Telegraph Co., Tokyo*, (1978).

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