Vol.2020-ARC-242 No.7
2020/10/12

複雑なデータ構造を持つアプリケーションを対象とした Approximate Memory 適応の検討

穐山 空道[1,a)]　塩谷 亮太[1]

概要：CPU 性能の増加に伴う DRAM アクセスレイテンシの相対的な増大が問題となっており、そのためデータに低確率でエラーが入ることを許す代わりに低レイテンシでのアクセスが可能な Approximate Memory が着目されている。本技術を用いるためにはアプリケーションの持つデータのうちエラー混入を許すデータを Approximate Memory 上に、エラー混入を許さないデータを通常のメモリ上に配置する必要がある。しかし一般にプログラムではエラー混入を許すデータと許さないデータが細粒度に混じって配置されるため、Approximate Memory 上に効率良く配置することができずなんらかの対策が必要である。例えば構造体により複数のデータをまとまて扱うアプリケーションでは、エラー混入を許すデータと許さないデータが入れ子になってメモリ上に配置され、前者のみを Approximate Memory 上に効率よく配置することは難しい。本稿では、まず SPEC CPU 2017 のソースコードとメモリアクセスパターンを分析し、このようなケースが実際に存在すること、データ構造を変換し許容されるエラー率が同一なデータを連続領域に集める手法ではアクセス局所性が低下しうることを示す。また解決法としてソフトウェアとハードウェアの協調動作によりプログラマの手間を少なく抑えつつアクセス局所性を低下させない手法を検討する。

1. 序論

CPU 性能の継続的な増加に対し、メモリのランダムアクセスレイテンシの相対的な増加が問題となっている。メモリアクセスがキャッシュミスする際のレイテンシは主にキャッシュミスであることが確定するまでの時間とその後 DRAM にアクセスする時間からなる。そのうち DRAM にアクセスする時間は DRAM 内のキャッシュを外した場合で最大 30 ns 程度であるが、この時間は 20 年以上ほぼ一定であり [1,2]、DRAM アクセスレイテンシのコンピュータ性能に対する重要性は増大している。

DRAM アクセスレイテンシを削減する技術の一つとして、Approximate Memory が着目されている。Approximate Memory とはメモリ内のデータにエラーが混入する確率が上がる代わり高速アクセスを実現するメモリであり、通常の DRAM のタイミング制約を仕様から意図的に逸脱させることで実現できる [2–4]。例えば [2] では、DRAM に送信する `ACT` コマンドの後に待機すべき時間を仕様の 12.5 ns から 7.5 ns に削減しても多くのビットが正しく読み出せることが示されている。

Approximate Memory を用い意味のある計算をするためには値が変わっても有意義な結果が得られるデータのみにエラーが入るよう制御する必要があるが、エラー率をビットごとに設定することは本質的に難しい。これはメモリのスループット向上のために多くのビットが電気的に同時に駆動され、タイミングの設定が多くのビットで共有されるからである。

前述の制限により、多くの実用プログラムでは Approximate Memory を直接適用することが難しい。例えばグラフ処理を行うプログラムがノードの評価値と次のノードへのポインタを構造体で管理する場合、ノードの評価値のみにエラーを入れることを許すためには数バイトごとにエラー率を設定する必要がある。本稿では構造体の中にエラー混入してもよいデータと混入してはいけないデータが混在しているアプリケーションを「複雑なデータ構造を持つアプリケーション」と呼び、このようなアプリケーションについて以下を検証する。

(1) 複雑なデータ構造を持つアプリケーションが実際にどの程度存在するかをソースコード分析で明らかにする。
(2) (1) で発見したアプリケーションに対し構造体のメンバ同士のアクセス共起度を計測する。アクセス共起度とは 2 つのメンバに対するアクセスの時間的近さを表す指標である。構造体の配列を各メンバごとの配列に分離するプログラム変換技術により複雑なデータ構造

[1] 東京大学 情報理工学系研究科 創造情報学専攻
Dept. of Creative Informatics, The University of Tokyo
[a)] akiyama@ci.i.u-tokyo.ac.jp

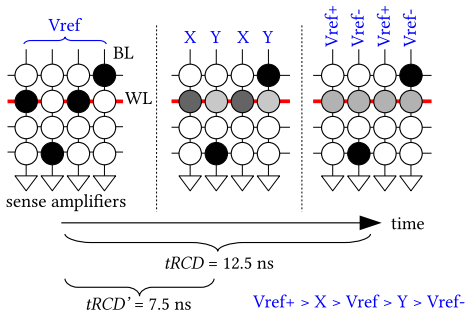

図 1 ACT コマンドの動作：読み出す row の WL を enable にした後 12.5 ns で BL の電圧が Vref+ または Vref- になることが保証される。Approximate Memory ではこれを縮めることでエラー率向上の代わり高速アクセスを達成する。

を持つアプリケーションにも Approximate Memory 適用できるが、アクセス共起度の高いメンバ同士を分離するとキャッシュミスの増加により性能上不利なケースがある。そこで実際のアプリケーションでメンバ間のアクセス共起度を計測する。

(3) アクセス共起度の強いメンバ同士を実際にメモリ上の離れた位置に配置した場合の性能変化を明らかにする。アクセス共起度の高いメンバ同士を分離しても、それぞれのメンバへのアクセスが独立にプリフェッチできる場合やアクセスが少ない（cold な）メンバとアクセスが多い（hot な）メンバを分離する場合には性能に影響しない。そこでシミュレータによりプログラム構造の変換を再現しこの性能変化を計測する。

(4) 複雑なデータ構造を持つアプリケーションのアクセス共起度を考慮し Approximate Memory を適応可能にする手法を予備検討する。

2. 背景

2.1 Approximate Memory の実現方式

Approximate Memory は DRAM の各コマンド発行後に設定される次のコマンドまでの待ち時間を調整することで実現できる。DRAM のコマンドとはメモリコントローラから DRAM に送信される信号であり、ACT、REF、READ などがある。特定の 2 つのコマンド間には最低限待つ必要がある時間や逆に特定の期間内に発行する必要があるなどのタイミング制約が仕様 [5, 6] で定まっている。例えば ACT コマンドと READ コマンドの間には最低 tRCD の間隔をあける必要があり、2 つの REF コマンドは tREF 以上の間隔をあけてはならない。しかしこの仕様はある程度の余裕を持って定められており、仕様を破っても多くのメモリセルが正常に動作することが報告されている。例えば [2] では tRCD を削減した場合のエラー率を計測し、[3, 4] では REF コマンドの間隔を tREF 以上にしてもエラーが起きない手法を提案している。

Approximate Memory では前述のタイミング制約を意図的に破りエラー率向上と引き換えに効率的な動作を実現する。ここでは DRAM 内のバッファに読み出したいデータをコピーするために発行する ACT コマンドの動作を例に Approximate Memory の実現方式を説明する。**図 1** は ACT コマンド実行時の DRAM の内部動作を示す。DRAM 内ではメモリセルと呼ばれるキャパシタが配列状に並び、ワードライン（WL）とビットライン（BL）で接続されている [7]。図の丸はメモリセルを表し、黒丸は電荷がチャージされていること、白丸は電荷がチャージされていないこと、灰色の丸は中間状態にあることを示す。図の左側は定常状態で、全ての BL はリファレンス電圧（Vref）にリセットされている。ここで ACT コマンドを発行すると、コマンドの引数である読み出し対象の行（row）が選択され、対応する WL が enable になる。図では上から 2 つ目の row を選択したものとし、当該 row 内のメモリセルが BL に接続される。すると接続されたメモリセルの電荷により BL の電圧が変化し、図の右側のように 1 を表す電圧 Vref+ または 0 を表す電圧 Vref- に達する。またこの時メモリセルでは BL への電荷の流出および BL からの電荷の流入が起こり、各メモリセルの電荷量は不安定（灰色の丸）になる。その後 BL の電圧を sense amplifier が増幅することでデータを読み出す。

Approximate Memory は ACT コマンドの後の待ち時間（tRCD）を削減することでエラー率向上と引き換えに高速アクセスを実現する。tRCD は DDR3-1600J の仕様 [5] では 12.5 ns と定義されている[*1]。しかし既存研究では tRCD を 7.5 ns まで削減してもほとんどのセルで正しい値が読み出せることが知られている [2]。tRCD を 7.5 ns に削減すると図 1 の中央のように BL の電圧が Vref+ や Vref- まで到達せず不安定な値になるため、値が正しく読み出せるケースと読み出せないケースが発生するが、ACT コマンドを高速に完了できるためメモリの性能が向上する。

2.2 Approximation の粒度

Approximate Memory において異なるエラー率を設定できる最小のデータ長を「Approximation の粒度」と本稿では呼ぶ。この粒度は DRAM の row サイズまでしか小さくできず、一般的に row サイズは 4 KB から 8 KB 程度であるため Approximation の粒度も 4 KB となる。第 2.1 章に示したようにコマンド間の待ち時間（tRCD など）は DRAM の row ごとに設定される。従って待ち時間を短くすることも row ごとにしかできず、Approximation の粒度は row サイズで制限される。

DRAM の row のサイズを 4 KB よりも小さくすることは性能と回路規模の両面から難しい。性能について、DRAM の動作速度は CPU が単位時間あたりに要求しう

[*1] その他の DDR3 や DDR4 の仕様についても近い値である。

るデータ量に対し極めて遅いため、多くのメモリセルを同時に駆動する必要がある。また回路規模についても少数のメモリセルごとに駆動するためには多くのトランジスタが必要となり現実的ではない。従って Approximation の粒度を数バイトなどに小さくすることは本質的に難しく、現実的なオーバーヘッドでは不可能である。

2.3 複雑なデータ構造を持つアプリケーションへの適応

本稿では構造体の中にエラー混入を許すデータとエラー混入を許さないデータが混在しているアプリケーションを「複雑なデータ構造を持つアプリケーション」と呼ぶ。例えば**図 2** のデータ構造を扱うプログラムにおいて、`struct node` のメンバ `score` は大小を比較しかつ多少の間違いがあっても許容される、すなわちエラー混入を許すデータであるとする。一方 `right` と `left` はポインタでありエラー混入を許さないデータである。このときこのデータ構造を扱うアプリケーションは複雑なデータ構造を持つアプリケーションと呼ばれる。

```
1 struct node {
2   struct node *right;
3   struct node *left;
4   double score;
5 } nodes[1024];
```

図 2 複雑なデータ構造の例：エラー混入してもよいデータ（score）とエラー混入してはいけないデータ（right、left）が構造体内に混在している。

複雑なデータ構造を持つアプリケーションではエラー混入を許すデータと許さないデータがメモリ上入れ子になるため、Approximate Memory をそのまま適応することはできない。図 2 では配列 `nodes` の指すメモリ領域内にメンバ `score` とそれ以外が入れ子になっている。従ってエラー率の設定は各構造体のインスタンスの 0 から 15 バイト目をエラーなし、16 から 23 バイト目をエラーありとすべきである[*2]。一方、第 2.2 章で見たように Approximation の粒度は 4 KB であるためこのような細かなエラー率の設定をそのまま実現することはできない。

3. ソースコード分析

3.1 分析方法

本章では複雑なデータ構造に多くのメモリアクセスを行うアプリケーションが実際に存在すること明らかにする[*3]。まずあるアプリケーションの扱うデータの中で最も多くのキャッシュミスを発生させるものを発見する。次にそのデータの型が C の構造体または C++のクラスであれば、そのアプリケーションは「複雑なデータ構造を持つ」と判断する。最も多くのキャッシュミスを発生させるデータに着目する理由は、そのようなデータが Approximate Memory の恩恵を受けやすいからである。逆にキャッシュヒット率が高くメモリからフェッチすることが少ないデータは Approximate Memory 上に置く利点が少ないため考慮しない。また複雑なデータ構造かどうかは定義に拠れば構造体またはクラスの中にエラー混入を許すデータと許さないデータの両方が入っていることを判定基準にすべきである。しかし構造体またはクラスのあるメンバがエラー混入を許すかどうかはアプリケーションの専門家でないと分からず、また同一のアプリケーションでもユースケースによって変わりうる。よって本稿ではエラー混入可能かどうかは判断せずメモリがアクセスが多発するデータが構造体またはクラスであるものを広く扱う。

あるベンチマークの中でキャッシュミスを最も多く発生させるデータは以下のように発見する。

(1) パフォーマンスカウンタにより各ベンチマークの LLC ミス率を計測し、20% 以上のものを選定する。キャッシュミス率が低いベンチマークは Approximate Memory の恩恵を受けにくいため除外する。
(2) Intel PEBS を用い、各命令ごとにキャッシュミス発生数の全体に対する割合を計測する。
(3) 全命令中で最も多くキャッシュミスを発生させている命令に対し、その命令がアクセスしているソースコード上の変数を発見する。

手順 (2) で用いる PEBS はパフォーマンスカウンタを全てハードウェア動作するよう拡張したもので、キャッシュミスの発生と記録の間の時間差が少なく命令レベルの計測が可能である。また手順 (3) で任意のバイナリを C/C++ コードに逆アセンブルすることは難しいため、バイナリに付随したデバッグ情報を参考に人手で分析を行う。

表 1 分析結果：左からベンチマーク名、LLC ミス率、最も多くキャッシュミスを発生させる命令がアクセスするデータの型、その型が構造体またはクラスであるかを示す。

ベンチマーク	LLC ミス率	LLC ミスを最も多く発生させるデータの型	構造体 or クラスか
deepsjeng	77.5 %	`ttentrty_t[]`	Yes
nab	64.9 %	`INT_T[]`	No
omnetpp	56.1 %	`sVector`	Yes
namd	50.4 %	`CompAtom[]`	Yes
lbm	48.8 %	`LBM_Grid (double[])`	No
x264	47.3 %	`uint8_t[]`	No
mcf	43.5 %	`arc_t[]`	Yes
gcc	36.6 %	-	-
blender	35.0 %	`VlakRen[]`	Yes
xz	31.6 %	`uint8_t[]`, `uint32_t[]`	No, No
perlbench	21.4 %	`char[]`	No

3.2 分析結果

分析結果を**表 1** に示す。対象のベンチマークは SPEC

[*2] padding はなく構造体は 24 バイトであるとする。
[*3] 本章は我々の先行研究 [8] と同じ手法を SPEC CPU 2017 に適用したものである。手法の詳細は [8] を参照のこと。

```
1  struct {
2    double x;
3    double y;
4  } points[1024];
5
6  for(i = 0; i<1024; i++) {
7    points[i].x = random(0, 1);
8    points[i].y = random(0, 1);
9  }
10
11 double center_x = 0, center_y = 0;
12 for(i = 0; i<1024; i++) {
13   center_x += points[i].x / 1024;
14   center_y += points[i].y / 1024;
15 }
```

図 3 ランダムに生成した点の幾何学的中心を求めるプログラム：データは構造体の配列で管理される。

```
1  struct {
2    double x[1024];
3    double y[1024];
4  } points;
5
6  for(i = 0; i<1024; i++) {
7    points.x[i] = random(0, 1);
8    points.y[i] = random(0, 1);
9  }
10
11 double center_x = 0, center_y = 0;
12 for(i = 0; i<1024; i++) {
13   center_x += points.x[i] / 1024;
14   center_y += points.y[i] / 1024;
15 }
```

図 4 AoS から SoA への変換例：図 3 のプログラムの実行結果を変えずにデータ構造を配列の構造体に変換した。

CPU 2017 のうち C または C++ で書かれたものとし、入力データは最大の `refrate` を用いた。また実行したマシンの CPU は Intel Xeon Silver 4108 で LLC サイズは 11 MiB である。表は左からベンチマーク名、LLC ミス率、当該ベンチマーク内で最もキャッシュミスを多く発生させるデータの型、そのデータ構造が構造体またはクラスであるかを示す。データ型の `[]` は配列であることを表し、`omnetpp` 以外の全てのベンチマークで LLC ミスを最も多く発生させるデータは配列である。なお `gcc` ベンチマークでは LLC ミスが多くの命令に分散しており特定のデータが多くのキャッシュミスを発生させていると言えないため結果を示していない。また `xz` では 2 つの命令が多くのキャッシュミスをほぼ均等に発生させているため、それぞれの命令に対応する結果を示す。

表より、キャッシュミス率が 20% 以上である SPEC CPU 2017 ベンチマーク 11 個のうち 5 個が複雑なデータ構造を持つ。該当するベンチマークは `deepsjeng`、`omnetpp`、`namd`、`mcf`、`blender` であり、それぞれチェス AI、離散イベントシミューレーション、分子動力学法シミューレーション、整数最適化問題、画像処理のベンチマークである。本結果より本稿で対象とする複数なデータ構造を持つプログラムが実際に多く存在することが明らかになった。

4. ソースコード変換とアクセス共起度

本章では複雑なデータ構造を持つアプリケーションを複雑なデータ構造を持たないよう変換する方法を述べ、Approximate Memory 適応の観点からその手法のデメリットを定量的に調査する。

4.1 AoS から SoA への変換

複雑なデータ構造を持つアプリケーションは、AoS（Array of Structures、構造体の配列）から SoA（Structure of Arrays、配列の構造体）への変換により複雑なデータ構造を持たないよう変換できる。これはプログラムの実行結果を変えないままメモリ上のデータ配置を変換するものである。例えば**図 3** は点 (x, y) の集合を構造体の配列 `points[1024]` で管理し、全ての点の中心を求めるプログラムである。なお `random(0, 1)` は 0 以上 1 以下の倍精度浮動小数点数をランダムに返す。このソースコードを**図 4** のように、すなわち配列 `x[1024]` と `y[1024]` の構造体を `points` とするよう書き換えると、実行結果を変えないままデータ構造を AoS から SoA に変換できる。

AoS から SoA への変換は構造体の配列から構造体の各メンバを取り出して独立した配列にするため、複雑なデータ構造を持つアプリケーションに Approximate Memory を適用可能にする。図 3 ではそれぞれの構造体のメンバ `x` と `y` は入れ子になってメモリ上に配置されるが、図 4 では構造体内に `x[1024]` と `y[1024]` が並んでおり `x[1024]` と `y[1024]` はそれぞれ連続したメモリ領域に配置される。このプログラムに Approximate Memory を適用し `x` のみにエラーが混入することを許すケースを考えると、図 3 では `x` のみを Approximate 領域に配置できないが、図 4 では `x` が Approximation の粒度より大きい連続領域に集まっているため配置可能である。

AoS から SoA への変換にはアクセス局所性を悪化させるケースがあり、Approximate Memory による高速化を妨げる可能性がある。例えば 図 3 では `points[i].x` と `points[i].y` は同じキャッシュラインに乗ることが期待されるが、図 4 では `points.x[i]` と `points.y[i]` はメモリ上の離れた位置にあるため同じキャッシュラインに乗らない。この例では `i` が 1 ずつインクリメントされるためプリフェッチャの効果によりキャッシュのデマンドミス数は変わらないが、アクセス順序がランダムであれば図 3 のデータ構造を図 4 のように変換するとデマンドミス数が 2 倍になると予想される。キャッシュのデマンドミス数の増加は実行速度の低下に直結し、このような場合には Approximate Memory による高速化アクセスのメリットを打ち消す可能性がある。

分析対象の構造体
```
struct S {
  int a;
  float b;
  char *c;
}
```
オフセット
```
S.a: 0, S.b: 4, S.c: 8
```
ヒープ上の S のインスタンス
```
0x1280000,
0x1280010,
...
```

Intel PIN の出力
```
1: 0x1280000, 4
2: 0x1280004, 8
3: 0x7ff1234, 8
4: 0x1280018, 8
5: 0x7fff432, 16
6: 0x1280008, 8
```
変換 ⇩
```
1: S.a, 0x1280000
2: S.b, 0x1280000
3: 0x7ff1234, 8
4: S.c, 0x1280010
5: 0x7fff432, 16
6: S.c, 0x1280000
```

図 5 メモリトレースの例：S へのアクセスはメンバ名とアクセス先の S のインスタンスの先頭アドレスのペアに変換する。

4.2 アクセス共起度分析：手法

そこで本稿では、第 3 章の分析で複数なデータ構造を持つと判定されたベンチマークについて、複数なデータ構造内のメンバ同士のアクセス共起度を調査する。ここで構造体 S の 2 つのメンバ S.a と S.b の「アクセス共起度」を、S.a へのアクセスが 1 つ与えられたときそのアクセスから過去に遡って近い時点に S.b へのアクセスが存在する確率と定義する。この値は既存の AoS から SoA への変換の研究 [9,10] において変換により得られる利得を変換前に推定するために使う手法に基づく。

構造体 S のメンバ間のアクセス共起度を計測するため、まず対象のベンチマークのメモリトレースを以下のように取得する。

(1) Intel PIN を用い、各メモリアクセスについてアクセスされたアドレスとアクセスサイズを記録する。そのアクセスがキャッシュヒットであるかキャッシュミスであるかは考慮しない。
(2) ベンチマークの実行時情報から、ヒープ上に確保された S のインスタンスのアドレスを取得する。
(3) ベンチマークのコンパイル済みバイナリから、S の各メンバの先頭からのオフセットを取得する。
(4) 取得した S のアドレスとオフセットを用い、トレースに含まれるメモリアクセスをアクセスされた S のメンバ名に変換する。またアクセスされた S のインスタンスの先頭アドレスも付与する。S へのアクセスでないメモリアクセスに対しては何もしない。

トレース取得の例を**図 5** に示す。図では S は 3 つのメンバ S.a、S.b、S.c を持つ。Intel PIN の出力の各行に対し、それがどのメンバへのアクセスであるか、またアクセスされた S のインスタンスの先頭アドレスは何であるかを各メンバのオフセット情報と S のインスタンスのメモリ上の位置から判断する。例えば変換後の「S.a, 0x1280000」（図の 1 行目）は、このアクセスが S.a へのアクセスでありかつアクセスされた S のインスタンスの先頭アドレスが 0x1280000 であることを表す。

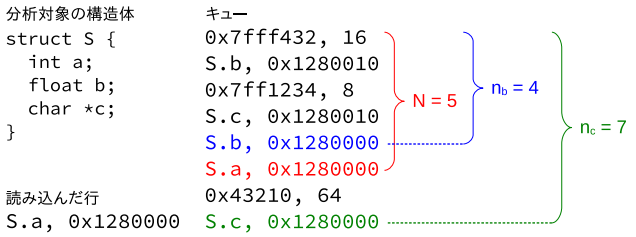

図 6 キューを用いた局所性計算の例

次に取得したトレースを用い S の各メンバ間のアクセス共起度を計算する。S の任意のメンバ x, y に関し、それらの間のアクセス共起度 $C_{x,y}$ は定義から「ある S.x へのアクセスが与えられたとき、そこから過去に遡って近い時点に S.y へのアクセスがある確率」である。これはトレースを先頭から順に読み全ての S.x へのアクセスについてその前何行かのトレースに S.y へのアクセスがあるかを見れば計算できる。ただし同じキャッシュラインや同じメンバに複数回アクセスがある場合にはトレースに複数の行があってもその間にキャッシュの状態は変わらないため、その効果を考慮する必要がある。

$C_{x,y}$ の具体的な計算手順を述べる。まず S の全てのメンバ x, y (x != y) について、$C_{x,y} = 0$ とする。次にトレースを先頭から順に読み込む。読み込んだ行が S へのアクセスではない場合、読み込んだ行をキューの先頭に入れる。キューの長さが閾値を越えた場合、キューの中に読み込んだ行と同じキャッシュラインへのアクセスがあればそれを削除し、なければキューの末尾の要素を削除する。読み込んだ行が S のメンバ（S.x とする）へのアクセスなとき、次を実行する。以下では**図 6** を用い説明する。

(1) キューに S.x へのアクセスでありかつアクセスされた S の先頭アドレスが読み込んだ行と同一のものがあれば、そのキュー内での位置を N とする（ただしキューの先頭を 0 番目とする）。そのようなアクセスがないとき $N = \infty$ とする。図では読み込んだ行がアクセスしているのは S.a、アクセスされた S の先頭アドレスは 0x1280000 であり、それと同じアクセスは $N = 5$ の位置にある。
(2) S.x 以外の全てのメンバ S.y について、キューに S.y へのアクセスでありかつアクセスされた S の先頭アドレスが読み込んだ行と同一のものがあれば、そのキューの先頭からの位置を n_y とする。そのようなアクセスが存在しないとき $n_y = \infty$ とする。図ではアクセスされた S の先頭アドレスが 0x1280000 であるような S.b へのアクセスと S.c へのアクセスがあり、$n_b = 4$、$n_c = 7$ である。
(3) S.x 以外の全てのメンバ S.y について、n_y が閾値 T 以下かつ N 未満ならば、$C_{x,y}$ をインクリメントする。いま $T = 256$ とすると、$n_b < N$ かつ $n_b < T$ な

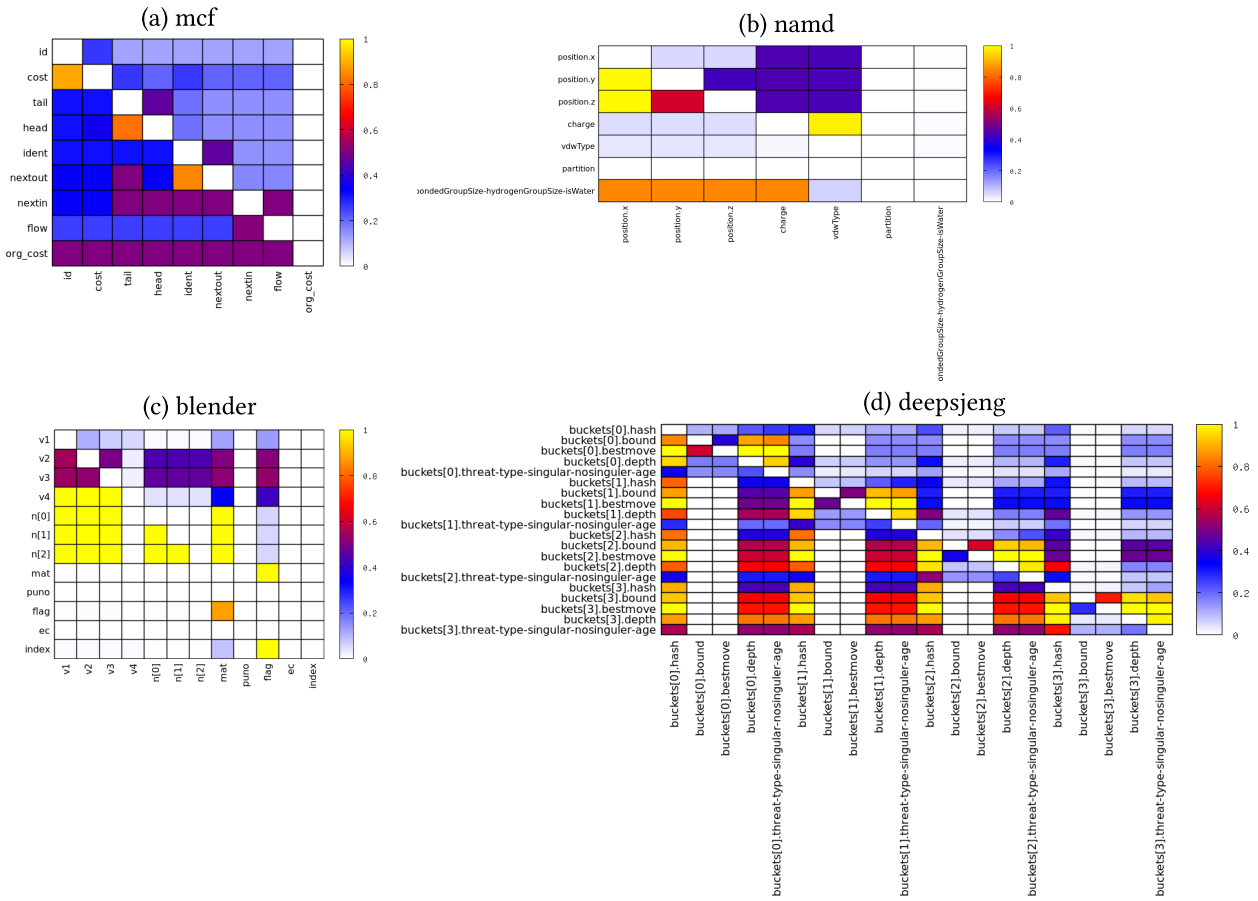

図 7 キャッシュミスを多く発生する構造体内の各メンバ間のアクセス局所性：それぞれの値は x 軸のメンバにアクセスがあった時に過去の近い時点に y 軸のメンバにアクセスがあった確率を表す。

ので $C_{a,b}$ はインクリメントされ、一方 $n_c > N$ なので $C_{a,c}$ はインクリメントされない。

(4) 読み込んだ行をキューの先頭に入れる。キューの長さが閾値を越えているとき、キュー内に `S.x` へのアクセスでありかつアクセスされた `S` の先頭アドレスが読み込んだ行と同一のものがあれば（すなわち $N \neq \infty$ ならば）それを削除する。そのようなアクセスがないときキューの末尾の要素を削除する。

以上をトレースの全ての行に対して実行後 $C_{x,y}$ をトレース内の `S.x` へのアクセス回数で割れば、$C_{x,y}$ は `x` と `y` のアクセス共起度になる。なお手順 (2) で $C_{a,c}$ をインクリメントしない意図は、この位置の `S.c` へのアクセスはキュー内の 5 番目の `S.a` へのアクセスをキューに入れた時にすでに考慮済みだからである。トレース内の同一のアクセスを複数回カウントしないためにキュー内で N より後にあるアクセスは考慮しない。

4.3 局所性分析：結果

図 7 に、第 3 章で複雑なデータ構造を持つと判断されたアプリケーションにおいてキャッシュミスを多く発生している構造体のメンバ間のアクセス共起度を示す。ただし `omnetpp` については 1 つのメンバへのアクセスしか検知できず、共起度が計算できないため結果を表示していない。各軸のラベルは構造体内のメンバ名であり、ヒートマップの値は横軸のメンバを `x`、縦軸のメンバを `y` としたときの $C_{x,y}$ を表す。構造体のメンバとして配列がある場合には配列の要素ごとに表示し、構造体のメンバとしてさらに構造体がある場合にはそのメンバを分割して表示している。例えば `blender` の `n[0]` は複雑なデータ構造の中にある `n` という配列の 0 番目の要素を表し、`deepsjeng` の `buckets[0].hash` は複雑なデータ構造の中の `buckets` という構造体の配列の 0 番目の要素の `hash` というメンバを表す。ダッシュ（`-`）で繋がれたメンバはビットフィールドであり、`a-b-c` は各フィールドの名前が `a`、`b`、`c` であることを表す。ビットフィールドは 1 バイトをさらに複数のフィールドに分けるが、メモリアクセスの最小単位は 1 バイトでありメモリアクセスの観点では各フィールドへのアクセスは区別できない。計算時のキューの長さは 4,096 とし、$C_{x,y}$ をインクリメントする閾値 T は 256 とした。

図より、複数なデータ構造を持つプログラムにおいてエラー混入を許すデータと許さないデータの間に強いアクセス共起度がある場合があることが分かる。例えば `mcf` の

id と cost は 0.8 程度のアクセス共起度を持つ。id はグラフの枝を一意に特定する識別子でありエラー混入してはいけないことが予測される。一方 cost はグラフの枝の評価値であり、このメンバにエラーが混入しても正しい結果を出力する場合があることは我々の先行研究 [11] で確認済みである。また blender の v1 から v3 と n[0] から n[2] は互いにアクセス共起度が強い。v1、v2、v3 は他の構造体を指すポインタであるためエラー混入を許さないデータであり、一方 n は float の配列であるためある程度のエラー混入を許すと予測される。このようなケースでは AoS から SoA での変換によってエラー混入を許すデータと許さないデータを分離するとキャッシュミスの増加により性能が悪化する可能性がある。

5. ソースコード変換の性能への影響

5.1 分析の内容と方法

これまでの分析により SPEC CPU 2017 に含まれるベンチマークに複雑なデータ構造を持つため Approximate Memory が単純には適応できないものがあること、それらのベンチマークの扱う構造体にアクセス共起度の高いメンバがあることが明らかになった。しかし互いにアクセス共起度が高い構造体のメンバ同士を AoS から SoA への変換でメモリ上の離れた位置に配置しても性能に変化がない、あるいは性能が向上する場合がある。要因として、それぞれのメンバのアクセスパターンがプリフェッチャにより予測可能な場合、アクセスが少ないメンバとアクセスが多いメンバが分離されることによるキャッシュヒット率の向上などがありえる。これら場合にはアクセス共起度の高いメンバをメモリ上の離れた位置に分離しても問題なく Approximate Memory の恩恵を受けられる。

そこで本章では AoS から SoA への変換を擬似的に実現することで、この変換によるメモリ配置の変化が性能に与える影響を調査する。AoS から SoA の変換は Approximate Memory 以外でもキャッシュヒット率の向上等の目的で研究されており、様々な既存研究が存在する [9,10]。しかし任意の C/C++ ソースコードを入力として AoS から SoA への変換を適応することは実装上容易ではない。特に C/C++ ではポインタが任意のアドレスを指せるため、変換対象の構造体内を指すポインタを発見する points-to analysis [12] が必要となる。

本稿では CPU シミュレータを改変しメモリアクセス命令のアクセス先アドレスを変換することで、構造体の中から指定したメンバを離れたメモリ上に配置し AoS から SoA への変換を擬似的に再現する。使用するシミュレータは gem5 のバージョン 20.0.0.0 である。**図 8** にシステムの動作を示す。

(1) ソースコード上で変換対象の構造体（図の S）の配列がヒープ上に確保される箇所を特定する。
(2) 特定した箇所にプリント文を挿入した後に通常の gem5 上で実行し、ヒープ上の S の配列の先頭アドレスとサイズを得る。
(3) 得たアドレスとサイズおよびメモリ上の離れた位置に配置するメンバの情報（図の remap info）を改変した gem5 に入力として与えベンチマークを実行する。remap info は各メンバのサイズ（図の size）とそのメンバを離れた位置に配置するかを表すブール値（図の remapped）を持つ。
(4) 改変した gem5 は remap info を元にアドレスを変換し、S の中の指定されたメンバ（図では v）のみをメモリ上の離れた位置に配置する。アドレスの変換はロードストアキューにリクエストが挿入される際に行う。

実行時情報から得られる S の配列の先頭アドレスは仮想アドレスである。gem5 ではロードストアキューに挿入されるのは仮想アドレスであり、かつロードストアキューよりも前に仮想アドレスから物理アドレスへの変換は行われないためこの方法で問題なく変換できる。また図の Memory layout にあるように S.v をメモリ上の離れた位置に配置すると同時にそれ以降のメンバを全て前方に移動する。例えば図では 0x40010 は S.id のアドレスだが、その前にある S.v が移動したため 8 バイト前に移動し 0x40008 に変換される。従って S の配列の 1 番目以降の要素では全てのメンバのアドレスが変更される。

本手法の利点は、実行時に仮想アドレスを変換することで複雑なソースコード変換を実際に行わずに変換の影響が見積もれることである。一方で欠点は実際に実行時にメモリ上で離れた位置に配置するメンバを指定する必要がある点である。このため全ての組み合わせを試すには 2 のメンバ数乗の施行回数が必要である。

5.2 分析環境とベンチマーク

表 2 シミュレートされる環境

ISA	x86_64
Issue Width	8 命令
Reorder Buffer	192 エントリー
L1 cache	16 KB + 16 KB, 2 way, 32 MSHRs
L2 cache	256 KB, 8 way, 32 MSHRs

AoS から SoA による擬似的な変換の性能への影響を、マイクロベンチマークおよび SPEC CPU 2017 を用いて調査する。マイクロベンチマークは構造体の中に 2 つの long 型（8 バイト）のメンバを持ち、その構造体の配列にアクセスしメンバの数値の和を求める。擬似的な変換では 2 つ目のメンバをメモリ上の離れた位置に配置する。SPEC CPU 2017 では第 4 章でアクセス共起度を計算したものの中から mcf、namd、deepsjeng を用いる。前述の通り分離する・しないについて全ての組み合わせを検証す

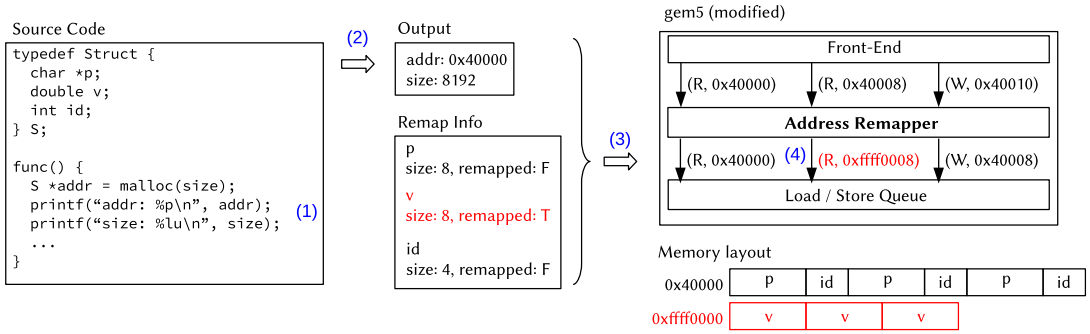

図 8 擬似的な AoS から SoA への変換の動作：S の各メンバへのアクセスをロードストアキューに入れる手前でアクセス先の仮想アドレスを変換する。

るには 2 のメンバ数乗回の実験が必要になるため、メンバを一つ選びメモリ上の離れた位置に配置する実験をメンバ数分行う。また SPEC CPU 2017 では初期化フェーズを各命令の機能のみ再現するモード（`AtomicSimpleCPU`）で実行し、その後キャッシュミス数などの統計情報をリセットして out-of-order 実行をサイクルレベルでシミューレーションするモード（`DerivO3CPU`）に切り替える。データセットには第 3 章と同じく最大のもの（`refrate`）を用い、out-of-order 実行に切り替えてから 7 千億サイクルをシミュレーションする。Out-of-order 実行モードの詳細なパラメータは**表 2** の通りである。

表 3 マイクロベンチマークの結果：sequential は構造体の配列に先頭から順にアクセスするケース、random はランダム順にアクセスするケースを表す。

		L1 miss	L2 miss
sequential	変換なし	1,050,859	133,326
	変換あり	1,050,715	133,187
random	変換なし	1,049,154	510,237
	変換あり	1,080,613	1,017,823

表 3 にマイクロベンチマークの結果を示す。表中の sequential は構造体の配列に連続アクセスする場合、random はランダムアクセスする場合を示す。L1 miss は gem5 の提供する `system.cpu.dcache.demand_misses` の値、L2 miss は `system.l2cache.demand_misses` の値である。なお実行時間や IPC は次にアクセスする構造体をランダムに選ぶ処理に時間がかかり大部分を占めると予想されるため示していない。

5.3 実験結果：マイクロベンチマーク

sequential ではアドレス変換を適応しても L1、L2 ともに miss 数が変わっていない。これは構造体の 2 つのメンバへのアクセスがそれぞれ独立のストリームとしてプリフェッチできることによると考えられ、このようなケースでは複雑なデータ構造を変換して Approximate Memory 上に一部のメンバのみを配置しても性能低下は起こらないと予想される。一方 random ではアドレス変換を適応すると L2 miss が約 2 倍に増加している。これは変換前では構造体の 1 番目のメンバにアクセスすると続く 2 番目のメンバへのアクセスはキャッシュヒットになるが、変換後では 2 つのメンバへのアクセスがそれぞれキャッシュミスを起こすためと考えられる。このようなケースでは複雑なデータ構造を変換して Approximate Memory 上に一部のメンバのみを配置すると性能に影響があると予想される。

5.4 実験結果：SPEC CPU

図 9 から**図 11** に、`mcf`、`namd`、`deepsjeng` に対して AoS から SoA への擬似的な変換を適用した際の性能指標を示す。横軸はメモリ上の離れた位置に配置する構造体のメンバを表す。ただし `namd` と `deepsjeng` に含まれるビットフィールドは見やすさのためメンバ名ではなく bitfields とする。また `deepsjeng` では複雑なデータ構造の中身自体が「メンバ数 5 の構造体の配列」であるが、配列の要素一つ一つを区別せず内側の構造体の 5 つのメンバについて 5 通りの実験を行う。各グラフの y 軸は変換を適用しない場合の値で正規化されており、値域に応じて異なるスケールを使用する。図の L1 miss は gem5 の提供する `system.cpu.dcache.demand_misses` の値、L2 miss は `system.l2cache.demand_misses` の値、ops は 7 千億サイクルの間にシミュレートできた micro operation の数であり、gem5 上では `sim_ops` の値である。

`mcf` では `cost`、`nextout`、`nextin`、`flow`、`org_cost` を分離すると L1 miss が増加（最大 9.0 %）し、逆に `cost`、`tail`、`head`、`ident` を分離すると L1 miss が減少（最大 −7.6 %）した。しかしいずれの場合にも L2 miss にはほとんど影響がなく、ops は L1 miss が増えたにもかかわらず増加しているケース（最大 1.4 %）がある。構造体のあるメンバを分離するとそれ以外のメンバは元よりも小さな領域にまとまるため、例えば独立に実行できないキャッシュミスが減って独立に実行できるキャッシュミス増えた（memory level parallelism が増加した）などの理由が考えられるが、今回は DRAM を詳細にシミュレートしていないためこれらの分析は今後の課題である。また `namd`

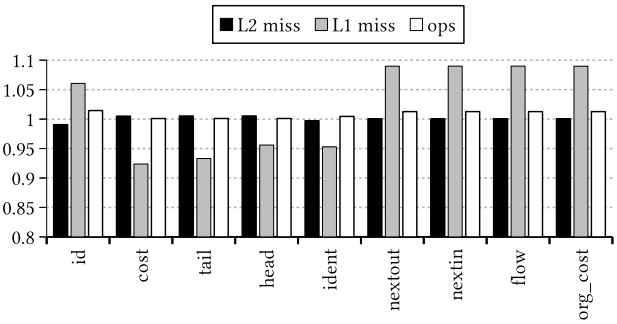

図 9 mcf への AoS から SoA への擬似的変換の影響

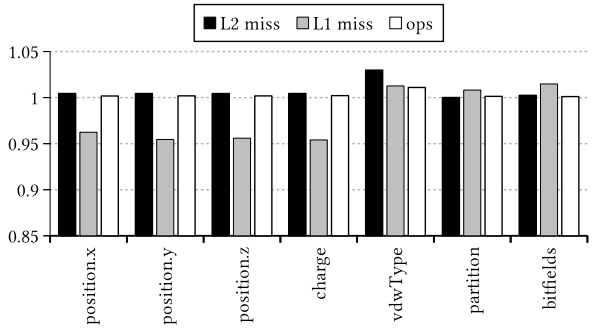

図 10 namd への AoS から SoA への擬似的変換の影響

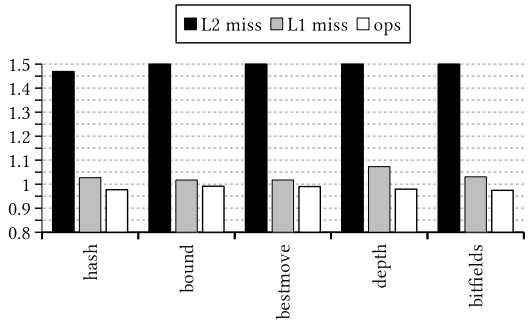

図 11 deepsjeng への AoS から SoA への擬似的変換の影響

でも L1 miss が最大 5 % 程度削減されているが、L2 miss や ops にはほとんど影響がない。一方 `deepsjeng` では L2 miss が全てのケースで約 1.5 倍に増加し、ops も最大 2.6 % 低下した。すなわち `deepsjeng` では構造体の特定のメンバを Approximate Memory 上に置くために AoS から SoA への変換を行うとキャッシュミスの増加により性能が低下し、Approximate Memory による恩恵が低減あるいはなくなることが予想される。

本実験は Approximate Memory を適用するために AoS から SoA への変換を適用した場合の性能変化を忠実に再現していると考えられる。この結果が第 4 章で計算したアクセス共起度から予測できれば、メモリトレースを一回操作するだけで全てのメンバに対しての実験結果を予測できることになる。しかし現状では前者と後者の間に明確な相関があるとは言えない。本実験で詳細が不明な点（例えばキャッシュミスが増えているにも関わらず高速化するケースがある理由など）を明らかにすることでメンバ間のアクセス共起度から本実験の結果を予測できるようにすることは今後の課題である。

```
1  typedef struct {
2    int x;
3    int y;
4  } pairs[N];
5
6  int ans = 0, index = 0;
7
8  for(i = 0; i < N; i++) {
9    if (fib(pairs[index].x) % 2 == 0)
10     ans += pairs[index].y;
11   else
12     ans -= pairs[index].y;
13
14   index = rand() % N;
15 }
```

図 12 Approximate Memory を単純に適応できないプログラム例：構造体のメンバ y のみをメモリ上の離れた場所に配置すると L2 miss が 2 倍になる。

6. 解決手法の検討

本章では本稿で検討してきた課題に対する解決策の検討内容を述べるが、詳細な実験などは今後の課題である。構造体内の特定のメンバのみを Approximate Memory 上に配置することによるキャッシュミスの増加は、通常の実行時に同一のキャッシュラインに乗る複数のメンバを同時にキャッシュにフェッチすることで解決できると考えられる。**図 12** は構造体の配列にランダムな順にアクセスするプログラムであり、`fib(n)` は n 番目のフィボナッチするを計算する関数であるとする。このプログラムに AoS から SoA への擬似的な変換を適応し、構造体のメンバ `y` のみをメモリ上で分離すると L2 miss に数が約 2 倍になる。しかし何らかの手法により `x` へのアクセスがあった際に `y` を同時にメモリからキャッシュにフェッチすることができれば、`fib(...)` の計算が終わり `y` にアクセスする時点では既に `y` がキャッシュに乗っておりメモリアクセスレイテンシを隠蔽できる。

本手法の実現には次の 2 点を解決する必要がある。

(1) どのメンバとどのメンバを同時にフェッチするかを何らかの方法で決定する
(2) 実際に 2 つのメンバの同時フェッチを何らかの方法で実現する

課題 (1) について、あるメンバ `x` にアクセスした時に別のメンバ `y` を同時にフェッチすることの利得は 2 つのメンバ間のアクセス共起度によって予測できると考えられる。共起度が弱ければ同時にフェッチしても `y` の値を使うまでの間にキャッシュから `y` が追い出されてしまう。逆に共起度が強すぎると同時にフェッチを開始しても `y` の値を使うまでにフェッチが終わらないためアクセスレイテンシを隠蔽できない。従ってこの課題を解決するためにはマシンの特性からどの程度のアクセス共起度であれば同時にフェッ

チすべきかを定めることが有効だと考えられる。

課題 (2) について、同時フェッチの実現には以下の 3 通りの方式が考えられる。

- 完全ソフトウェア方式：AoS から SoA への変換をコンパイラなどで行い、適切な位置に y をプリフェッチする命令を追加で挿入する。
- 完全ハードウェア方式：AoS から SoA への変換はアドレス変換により擬似的に行い、y を適切な位置でハードウェアが追加の命令なしにプリフェッチする。
- ハイブリッド方式：AoS から SoA への変換はアドレス変換により擬似的に行い、適切な位置に y をプリフェッチする命令を追加で挿入する。

完全ソフトウェア方式では AoS から SoA へのコンパイラでの変換が必要になるが、これは前述の通りポインタが任意の位置をさせることから実装上難しい。実際 gcc に実装されていた構造体内のメンバをメモリ上で並び替える structure reordering 機能は、“not always work correctly” [13] との理由で削除され現在まで再実装されていない。また完全ハードウェア方式では追加命令なしにプリフェッチを行うため、ハードウェア内での整合性をとるための制御が必要である。例えば x とそれに対応する y の両方のアドレスがページフォルトを起こした場合、1 つの命令に対し例外が 2 つ発出されることになり、そのような例外のセマンティクスを新たに定義する必要がある。

これに対し、ハイブリッド方式では前述のような困難が発生しない。ハイブリッド方式では AoS から SoA への変換は本稿で行うように仮想アドレスを変換すればよく複雑な処理は不要である。仮想アドレスの変換を実際のハードウェアで行う際のオーバヘッドについて、回路面積についてはごく少数の再配置情報を持つのみであるため非常に小さい。ただし変換にかかる時間がクリティカルパスにならないかどうかは実際に SPICE シミュレータなどで設計と検証が必要である。また y をプリフェッチする命令は単に通常の `prefetch` 命令を y の変換前のアドレスに対して発行すれば通常のメモリアクセスと全く同様に y の分離先のアドレスに変換される。またページフォルトについては y をプリフェッチする命令が実際に存在することから通常のページフォルトと同様に扱えばよく、例外に関してハードウェアを変更する必要はない。

7. 関連研究

Approximation の粒度に関して考察している研究は我々の知る限り Nguyen ら [14] によるものと我々の既存研究 [8,15] しか存在しない。Nguyen ら [14] は深層学習アプリケーションに限り Approximation の粒度の問題を解決している。浮動小数点数では上位ビットの方がエラーが実際の数値に与える影響が大きいため、ビットごとに異なるエラー率の設定が望ましい。そこでこの研究ではメモリ上のデータの行方向と列方向を入れ替えることでこれを実現する。通常の DRAM では連続データは同一の row 内に格納されるが、提案システムでは連続データは同一の column（図 1 で縦の方向）に格納される。この手法は浮動小数点数のビットごとに異なるエラー率の設定を可能とするが、32 bit の連続するデータを読み出すには 32 の row にアクセスする必要があり、深層学習のように巨大な行列をまとめて読み出すアプリケーションにしか適用できない。また我々の最初の既存研究 [15] では Approximation の粒度の問題についてはじめて言及したが、複雑なデータ構造を持つアプリケーションがどの程度程度存在するかや、ソースコード変換によるメモリ配置の変換については考察していない。そこで続く研究 [8] では複雑なデータ構造を持つアプリケーションの存在可能性を SPEC CPU 2006 およびグラフ分析アプリケーションを分析することで明らかにした。本稿はこれらをさらに発展させたものである。

データの特性に応じて異なるエラー率を設定する研究は広く行われている。Liu ら [16] は メモリセルに電荷をためなおす操作である `REF` コマンドの間隔を仕様より長くしエラー率を高める代わりに効率化を達成する。この研究ではメモリを `REF` コマンドの間隔が異なる bin に分けプログラマが指定したデータの重要度に応じて格納する bin を決定する。しかし `REF` コマンドも `ACT` コマンドと同様 row 単位でメモリセルを駆動するため、本稿と同じ Approximation の粒度の問題が発生する。Raha ら [17] はこれを発展させ、`REF` コマンドの間隔を広げた際のエラー特性（エラー率、エラー発生位置など）を計測する[*4]ことでエラー率の詳細な制御を可能にした。しかし本研究でもコマンドの間隔は row ごとにしか設定できず、また計測されたエラー率をそのまま使うよりないため Approximation の粒度の問題は解決できない。また Chen ら [18] は DRAM の bank ごとにエラー率を設定し、データの重要度に応じ割り当てる bank を変更するメモリコントローラを提案した。bank は DRAM のチップ内にあるメモリセルのグループのことであるが、これは row よりもさらに大きく通常 256 MB から 1 GB 程度のサイズある。従って bank ごとのエラー率設定では複雑なデータ構造を持つアプリケーションをそのまま実行することはできない。

謝辞 本研究は、JST、ACT-I、JPMJPR18U1 の支援を受けたものである。

参考文献

[1] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA,

[*4] エラー発生は製造ばらつきやベンダーごとの差異に大きく影響されることが知られており、コマンドの間隔が同じでもビットエラー率が同じとは限らない。

USA (2006).
[2] Chang, K. K., Kashyap, A., Hassan, H., Ghose, S., Hsieh, K., Lee, D., Li, T., Pekhimenko, G., Khan, S. and Mutlu, O.: Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization, *International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)*, pp. 323–336 (2016).
[3] Hassan, H., Pekhimenko, G., Vijaykumar, N., Seshadri, V., Lee, D., Ergin, O. and Mutlu, O.: ChargeCache: Reducing DRAM latency by exploiting row access locality, *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 581–593 (2016).
[4] Zhang, X., Zhang, Y., Childers, B. R. and Yang, J.: Restore truncation for performance improvement in future DRAM systems, *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 543–554 (2016).
[5] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION: JEDEC STANDARD: DDR3 SDRAM Standard, JESD79-3F (2010).
[6] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION: JEDEC STANDARD: DDR4 SDRAM Standard, JESD79-4B (2013).
[7] Jacob, B., Ng, S. and Wang, D.: *Memory Systems: Cache, DRAM, Disk*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007).
[8] Akiyama, S.: Assessing Impact of Data Partitioning for Approximate Memory in C/C++ Code, *The 10th Workshop on Systems for Post-Moore Architectures (SPMA)*, pp. 1 – 7 (2020).
[9] Ye, L., Lis, M. and Fedorova, A.: A Unifying Abstraction for Data Structure Splicing, *International Symposium on Memory Systems (MemSys)*, p. 173–183 (2019).
[10] Zhong, Y., Orlovich, M., Shen, X. and Ding, C.: Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity, pp. 255 – 266 (2004).
[11] Akiyama, S.: A Lightweight Method to Evaluate Effect of Approximate Memory with Hardware Performance Monitors, *IEICE Transactions on Information and Systems*, Vol. E102-D, No. 12, pp. 2354–2365 (2019).
[12] Steensgaard, B.: Points-to Analysis in Almost Linear Time, *Symposium on Principles of Programming Languages (POPL)*, p. 32–41 (1996).
[13] Free Software Foundation, Inc: GCC 4.8 Release Series Changes, New Features, and Fixes, https://gcc.gnu.org/gcc-4.8/changes.html (2019).
[14] Nguyen, D. T., Hung, N. H., Kim, H. and Lee, H.-J.: An Approximate Memory Architecture for Energy Saving in Deep Learning Applications, *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–14 (2020).
[15] 穐山空道，塩谷亮太：Approximate Memory のデータ分離に起因する性能低下を抑制するプリフェッチ手法，組込み技術とネットワークに関するワークショップ (ETNET 2019)，pp. 1 – 10 (2019).
[16] Liu, S., Pattabiraman, K., Moscibroda, T. and Zorn, B. G.: Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning, *SIGARCH Comput. Archit. News*, Vol. 39, No. 1, pp. 213–224 (2011).
[17] Raha, A., Sutar, S., Jayakumar, H. and Raghunathan, V.: Quality Configurable Approximate DRAM, *IEEE Transactions on Computers*, Vol. 66, No. 7, pp. 1172–1187 (2017).
[18] Chen, Y., Yang, X., Qiao, F., Han, J., Wei, Q. and Yang, H.: A Multi-accuracy Level Approximate Memory Architecture Based on Data Significance Analysis, *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 385–390 (2016).