Vol.2019-ARC-235 No.18
Vol.2019-SLDM-187 No.18
Vol.2019-EMB-50 No.18
2019/3/17

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

穐山 空道[1,a)]　塩谷 亮太[1]

概要：CPU コアの性能向上に伴い、メモリアクセスレイテンシが相対的に増大している。またメインメモリの容量増大に伴い、メモリの消費電力がマシン全体の消費電力に占める割合も増大している。これらにより、メインメモリのレイテンシと消費電力はコンピュータ性能のボトルネックとなっている。この問題に対し、DRAM に保存されるデータに低確率でエラーが混入することを許す代わりに高速アクセスと消費電力削減を実現する Approximate Memory が有用である。しかし、Approximate Memory ではエラー混入を許すデータと許さないデータを分離して保持する必要があり、メモリアクセスの空間局所性悪化によりアプリケーションの性能が低下する。本稿では、まず Approximate Memory を想定したデータ分離によるアプリケーションの性能低下をシミュレーションにより定量的に分析する。次に、データ構造の先頭へのアクセス時にエラー混入を許すデータを同時にフェッチするすることで性能低下を抑制する手法を提案し、シミュレーションによって効果を検討する。

1. 序論

メインメモリの消費電力とアクセスレイテンシは、近年のコンピュータの性能を決める大きな要因である。機械学習やビッグデータ処理などのためにコンピュータのメモリ容量は増大しており、マシンの消費電力の 25% から 40% 程度がメモリに使われているという報告もある [1,2]。従ってメモリの消費電力はマシン全体の電力性能を決める上で重要な指標である。またプロセッサコアの性能増加が著しい事に反し DRAM のレイテンシはほぼ一定で推移しており [3,4]、近年ではメモリのアクセスレイテンシがより実行性能のボトルネックになり易い。

これらの課題に同時にアプローチする手法として、Approximate Memory が有用である。Approximate Memory は Approximate Computing の一種であり、メモリ上のデータに低い確率でエラーが混入することを許す代わりに高速アクセスや消費電力削減を実現できる。既存のデバイスレベルの研究では、DRAM の内部動作にかかる待ち時間を仕様より短縮するものがある [4–6]。これらの研究ではエラー率が向上する代わりに高速アクセス・低消費電力を実現できることが知られている。これらの手法をより積極的に適用することで Approximate Memory を実現できる。

Approximate Memory の利用では、プログラムコードやポインタなどのエラー混入を許さないデータ（保護データ）と、行列の値などのエラー混入を許すデータ（近似データ）を分離することで保護データの一貫性保証と近似データアクセスの高速化を両立できる。DRAM の構造上エラー混入率をビットごとに指定することは難しく、現実的には DRAM の row ごとなどの粒度でエラー混入率を変化させることが考えられる。しかし、row は 4K ビットや 8K ビットの大きな単位であり、row ごとの粒度で分離を実現するとアクセス局所性を悪化させアプリケーション性能を低下させる。例えば各ノードがポインタで繋がったグラフ構造上を探索するアプリケーションに対し、各ノードの評価値を近似データ、隣のノードへのポインタを保護データとする分離を考える。この時、通常のメモリ上では評価値と隣のノードへのポインタは同一キャッシュラインに存在する可能性が高く高速なアクセスが可能である。一方、保護データと近似データを分離した Approximate Memory 上では評価値と隣のノードへのポインタは DRAM 上で離れた場所に配置され同一キャッシュラインに乗らない。

本稿では、保護データと近似データの分離によって発生するアプリケーションの性能低下をシミュレーションによって評価し、性能低下を軽減するためのプリフェッチ手法を提案する。提案するプリフェッチ手法は、分離された保護データと近似データへのアクセスがソースコード上で近くにあることを利用する。メモリを確保する際に保護

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

データと近似データを関連付けて管理し、プログラマが保護データにアクセスする際に専用のインターフェースを用いることで、保護データにアクセスした際に同時にその近傍の近似データをフェッチする。シミュレーションによる評価の結果、提案手法が最も効果的な場合にはデータ分離による性能低下を完全に抑制できることが明らかになった。一方、データサイズによっては性能低下を全く抑制できない場合もあった。また、ワークロードによってはデータ分離がクリティカルパスを決定するデータとそうでないデータを分離するためデータ分離によって性能が向上するケースもあり、この効果の考慮は今後の課題である。

2. 背景

2.1 Approximate Memory

メインメモリの消費電力およびレイテンシは近年のコンピュータの重要な性能決定要因である。第一に、機械学習やビッグデータ処理など大量のデータを扱うワークロードの隆盛によりメインメモリ容量は増大し続けている。Intel Xeon Platinum 8160 M プロセッサでは 1 ソケットあたり最大 1.5 TB のメモリを搭載でき、また NVIDIA DGX-2 は 2 ソケットで 1.5 TB のメモリを持つ。メモリ容量の増大により、マシン消費電力の 25 % 〜 40 % がメモリに消費されるという報告もある [1,2]。第二に、CPU コアの著しい性能向上に反し、メインメモリのアクセスレイテンシは過去 20 年に亘りほぼ一定である [3,4]。理由として、(1) レイテンシは帯域と異なり並列化で改善できないこと、(2) DRAM の製造プロセスを縮小しても電気的動作にかかる時間は削減できないこと、(3) メモリ帯域向上のためにキャッシュ階層を追加するとレイテンシが悪化することが挙げられる。結果として、CPU コア性能を基準とした相対的なメモリレイテンシは著しく悪化している。

DRAM の消費電力とレイテンシを同時に削減する手法として、Approximate Memory が有用である。Approximate Memory はコンピュータの正確な動作を犠牲にする代わりに高速アクセスや低消費電力を実現する Approximate Computing の一種である。Approximate Memory は DRAM 内のデータにエラーが混入することを許す代わりに低レイテンシ、低消費電力を実現する。機械学習やビッグデータ処理などのワークロードではデータに元々ノイズが含まれていること、計算に要求される精度が高くないことから特に有用だと考えられる。

Approximate Memory を実現するために、DRAM の電気的動作の待ち時間を仕様よりも短くし高速動作と省電力を得る既存手法の適用が考えられる。DRAM の電気的動作には activation、restoration、precharge 等があり [7]、それぞれ動作開始から別の動作が実行可能になるまでの時間が定められている。この待ち時間は製造上のブレやマージンなどを考慮した最悪値が定められており、既存研究ではこのマージンを削減することが提案されている。例えば (1) activation と precharge の待ち時間削減とエラー混入率の関係を実際の DRAM で計測した研究 [4]、(2) 近い将来アクセスされる row を予測し当該 row の restoration の待ち時間を削減する研究 [6]、(3) 最近アクセスされた row には多くの電荷が存在することを利用し activation の待ち時間を削減する研究 [5] などがある。これらの研究はデータにエラーが混入しない程度に待ち時間の削減率を抑えることを提案するが、待ち時間削減量をより大きくすれば Approximate Memory を実現できる。

2.2 Approximate Memory のためのデータ分離

Approximate Memory をアプリケーションから利用するには、エラー混入を許すデータ (「近似データ」と呼ぶ) とエラー混入を許さないデータ (「保護データ」と呼ぶ) を区別し、異なるエラー率を設定することが一般的である。既存研究では DRAM 内のブロックをエラー耐性ごとの bin に分け、保護データをエラー耐性の高い bin に、近似データをエラー耐性の低い bin に配置する [8] [9] [10]。プログラムのバイナリコードやポインタなどは保護データとしてエラーが混入しないことを保証する必要があり、一方で数値計算に用いる行列の値などは近似データとして高速・省電力なアクセスを実現する必要がある。本稿では近似データと保護データの分離を「データ分離」と呼ぶ。

データ分離の議論には、一般に DRAM 上にデータがどのように配置されるかの考慮が必要がある。DRAM はキャパシタから構成され、それらのキャパシタは column、row、bank、rank、channel の階層構造になっている。DRAM の内部構造については文献 [7] が詳しい。ある物理アドレスを持つデータをどの column、row、bank、rank、channel に配置するかを決定する方法を「アドレスマッピング」と呼ぶ。例えばあるマシンの row 数が 65536 ($= 2^{16}$) のとき、物理アドレスの上位 16 ビットでそのアドレスに対応する row を決定できる。一般に、row の決定には上位のビットが、bank と channel の決定にはアドレスの中位から下位のビットが使用される。複数の bank や channel は並列動作できるため、データを多くの bank、channel に分散させて性能向上が図られる。一方、異なる row には同時にアクセスできずかつ row 切り替えには待ち時間を要するため、データを少数の row に集中させるために row は上位ビットで決定することが一般的である。

アドレスマッピングを考慮し、本稿ではデータ分離は DRAM の row ごとに異なるエラー率を設定し近似データと保護データを異なる row のセットに配置すると想定する。2 つの row に異なるエラー率を設定するとは、各 row に対して実行される activation 等の待ち時間に異なる値を用いることに対応し、メモリコントローラの改変で実現できる。前述のように一般に row は物理アドレスの上

Vol.2019-ARC-235 No.18
Vol.2019-SLDM-187 No.18
Vol.2019-EMB-50 No.18
2019/3/17

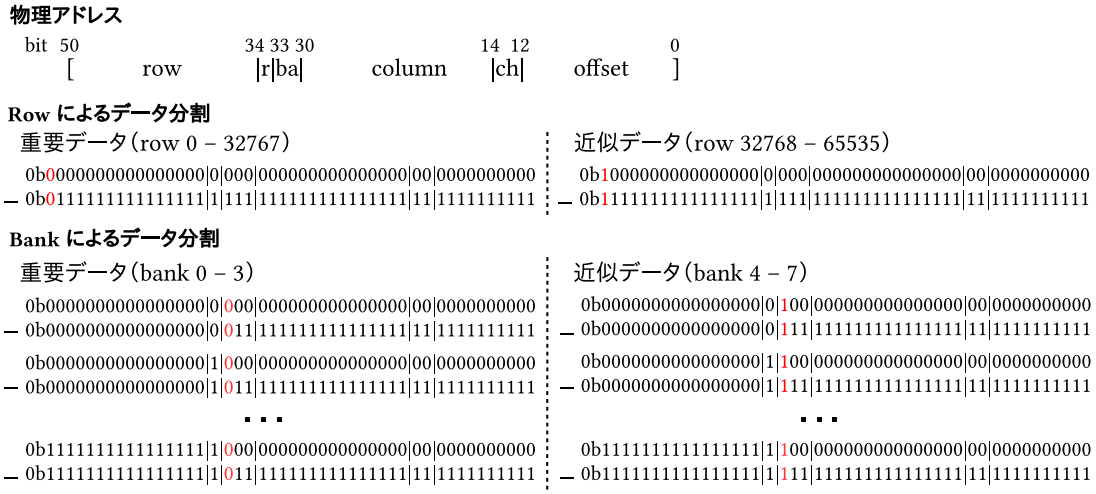

図 1 データ分割における保護データと近似データの割り当てられる物理アドレスの範囲。row はアドレスの上位ビットで決まるため、row を用いてメモリを 2 分割可能である。一方 bank はアドレスの中間のビットで決まるため bank を用いてメモリを 2 分割できない。

位のビットで決定されるため、row を用いた分割によって DRAM を 2 分割可能である。bank や channel ごとに異なるエラー率を設定することも可能だが、これらはアドレスの中位から下位ビットで決定されるため保護データと近似データが細かい領域に分割される。**図 1** は、row を用いてデータ分割を行う場合と bank を用いてデータ分割を行う場合の、保護データと近似データが配置される物理アドレスの範囲を示す。アドレスマッピングは gem5 シミュレータのデフォルトを示しており、上から row 、rank、bank、column、channel、page offset を決定する。row による分割では最上位ビットの値によって保護データと近似データが切り替わるため、保護データと近似データはそれぞれ 1 つの連続領域に配置される。一方、bank による分割では bank を決める 3 bit 中の最上位ビットによって保護データと近似データが切り替わるが、さらに上位のビットが存在するため保護データ、近似データはそれぞれ多数の分割された領域に配置される。

3. データ分離による性能低下の原因

本章では、データ分離によってアプリケーションの性能が低下することを定性的に説明する。定量的な評価は第 6.2 章で行う。Approximate Memory を利用するためのデータ分離は、元のソースコードでは連続配置されているデータを分離するため、メモリアクセスパターンの変化による性能低下を引き起こす。データ分離のパターンには、疎粒度と細粒度の 2 つが考えられる。「疎粒度データ分離」は、大きな連続領域全体をまとめて近似データとする場合に適用される。疎粒度データ分離ではメモリアクセスパターンは変化しないため、性能低下は発生しないと考えられる。「細粒度データ分離」は、例えば構造体内の一部のメンバのみを近似データとしたい場合に適用される。この場合、構

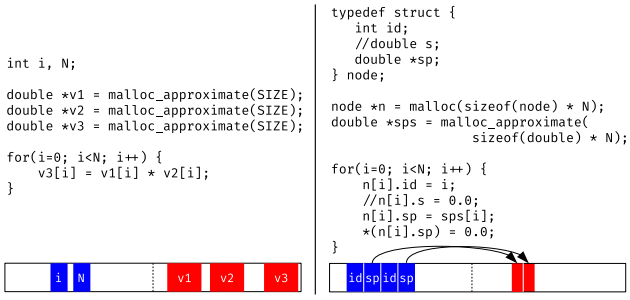

図 2 疎粒度データ分離の例（左）と細粒度データ分離の例（右）。疎粒度では大きな連続データをそのまま近似データとすればよい。一方細粒度では構造体内の一部のメンバのみを近似データとし、保護データとの対応付けを取る必要がある。

造体の領域のうち一部のみを近似データとして確保する必要がある。これにより細粒度データ分離では以下のような性能低下要因が考えられる。

(1) 連続していた構造体が分割され局所性が悪化することによるキャッシュミス増加
(2) 構造体の位置から対応する近似データの位置を知るためのメモリアクセス命令の増加

図 2 にデータ分離を実現するナイーブな方法の例を示す。図の左側が疎粒度データ分離、右側が細粒度データ分離である。近似データの確保には専用の関数である `malloc_approximate` を用い、通常の `malloc` で確保された領域やスタックなどは全て通常データとなるものとする。図の左側はベクトル積の計算で計算対象 `v1`, `v2` と結果 `v3` を近似データとする例である。メモリアクセスパターン、データへのアクセス方法ともに通常の `malloc` で `v1`, `v2`, `v3` を確保する場合と全く同じであり、データ分離によって性能は変化しないと予想される。

図の右側は、構造体 `node` の要素 `id`, `s` のうち、`s` のみを近似データとする例である。コメントアウトされて

Vol.2019-ARC-235 No.18
Vol.2019-SLDM-187 No.18
Vol.2019-EMB-50 No.18
2019/3/17

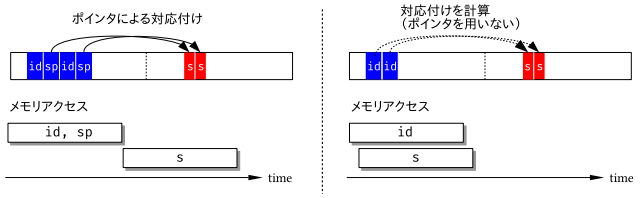

図 3 提案手法の概要。ナイーブな分離（左）では id と s へのアクセスは直列化されるが、提案手法（右）では s の位置をポインタを用いず計算するため id と s は同時にアクセスできる。

いる部分が通常のメモリ上でのプログラム、それ以外が Approximate Memory のためにデータ分離されたプログラムである。構造体内の s のみを近似データとして確保するため、構造体のメンバをポインタ sp に置き換え、近似データには sp を pointer dereference することでアクセスしている。データ分離されたプログラムでは、近似データ *sp へのメモリアクセスの前に近似データの位置を知るためのポインタ sp にアクセスする必要がある。また、sp は id と同一キャッシュラインに存在する可能性が高いが、*sp は物理メモリ上で離れた位置にあるため id と同時にキャッシュに載ることはない。さらに、sp の値が確定するまで *sp にはアクセスできないためこれらのメモリアクセス命令は直列化される。

4. 提案手法

4.1 手法の概要

本研究では、Approximate Memory を利用するための細粒度データ分離における性能低下を抑制する手法を提案する。基本アイディアは以下の二点である。

(1) 保護データと近似データは 1 対 1 に対応するため、ある保護データに対応する近似データの位置はポインタなしで決定できる。

(2) 対応する保護データと近似データは物理メモリ上離れているため、メモリレベル並列性によって同時にフェッチできる。

図 3 に提案手法の概要を示す。図の上段はメモリ内のデータ配置、下段はメモリアクセスの時系列を示し、図の左がナイーブな分離、右が提案手法を表す。保護データと近似データの対応は、ナイーブな分離ではポインタによって管理される。一方、提案手法は対応付けをアドレス計算によって求めるためポインタへのアクセスが不要である。このアドレス計算はポインタを用いた管理より低コストで実現できる。近似データの位置計算の詳細は第 4.2 章に示す。ナイーブな分離では近似データを指すポインタ（図の sp）の読み込みが終わるまで近似データの読み込みを開始できず、保護データと近似データの読み込みは直列化される。一方、提案手法では読み込む保護データの位置が決まると同時に対応する近似データの位置が決まる。従って、CPU は保護データと近似データへの load 命令を並列に発行でき、保護データと近似データはメモリ上で物理的に離れているため DRAM 内の並列性によって二つの load は DRAM 内で同時に処理される。即ち、提案手法は近似データを実際に利用する前にプリフェッチする。

4.2 近似データの位置計算

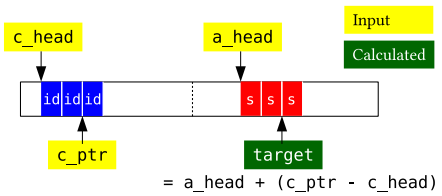

図 4 アクセサによる近似データ位置の計算方法。黄色が入力、緑色が計算する値。c_head、c_ptr、a_head は繰り返し参照されるためレジスタやキャッシュに載っている事が期待できる。

図 4 に、アクセスしようとする保護データの位置から対応する近似データの位置を計算する方法を示す。黄色の背景が入力、緑色の背景が計算する値である。c_head が保護データの先頭位置、c_ptr がアクセスしようとする保護データの位置、a_head が近似データの先頭位置である。c_head と a_head は本稿で実装するメモリアロケータによって管理されるため既知であり、c_ptr はアクセスしようとする保護データの位置であり当然既知である。

提案手法はポインタによるナイーブな分離よりもメモリアクセスコストが低く、データ分離しない場合と同様のコストで実現できる。図 4 で使われる変数のうち、c_head と a_head は全ての近似データアクセスに関して同一である。よってこれらの変数はレジスタやキャッシュに載っていることが期待でき、アクセスコストは無視できる。また c_ptr の読み出しと target への書き込みはデータ分離しない場合でも必要である。以上から提案手法のメモリアクセスコストはデータ分離しない場合と同様である。

4.3 効果の分析

提案手法の有効性を事前確認するため、プリフェッチ可能なメモリアクセスの割合及びその bank conflict 率を調査する。**図 5** は、近似データへの全アクセスのうちプリフェッチ可能なものの割合を示す。「プリフェッチ可能」とは、n 番目の保護データへのアクセスの後に n 番目の近似データへのアクセスが存在し、かつ二つのメモリアクセスの間に w 個未満のメモリアクセスしかない場合と定め、w をウィンドウサイズと呼ぶ。遠い将来に利用されるデータをプリフェッチすることはキャッシュ容量の圧迫につながりかつフェッチしたデータが利用される前にキャッシュから追い出される可能性も高いため、ウィンドウサイズが小さい場合のプリフェッチ可能率が高いことが望ましい。図 5 の横軸はウィンドウサイズ、縦軸は当該ウィンドウサイズ

Vol.2019-ARC-235 No.18
Vol.2019-SLDM-187 No.18
Vol.2019-EMB-50 No.18
2019/3/17

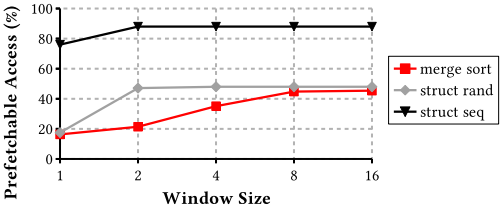

図 5 近似データへのアクセスのうちプリフェッチ可能なものの割合。保護データへのアクセスの後 Window Size 個のメモリアクセス命令のうちに対応する近似データへのアクセスがある場合を「プリフェッチ可能」とする。

でのプリフェッチ可能率を示す。$w = 16$ でのプリフェッチ可能率は `merge sort`、`struct rand`、`struct seq` でそれぞれ 46.0 %、48.0 % 、88.0 % である。

表 1 保護データと近似データの bank conflict 率 ($w = 16$)。ほぼ 12.5 % すなわち bank 数分の 1 になっている。

merge sort	struct seq	struct rand
12.5 %	12.6 %	12.4 %

表 1 に、プリフェッチ可能な近似データアクセスのうち、対応する保護データと同一の DRAM bank にあったものの割合（bank conflict 率）を示す。一つの bank 中では一つの row にしか同時にアクセスできないため、同一 bank 中の異なる row へのアクセスは直列化される。従って bank conflict が発生するとプリフェッチの効果が得られない。表より、bank conflict 率は 12.5 % ± 1% 程度である。これはシミュレートした DRAM の bank 数が 8 なことによると考えられる（$\frac{1}{8} \times 100 = 12.5$）。同一の bank へのアクセス集中にメリットはないため、物理アドレスから bank への対応付けはメモリアクセスがなるべく多くの bank に分散するように決定される。従って、より一般的な 16 bank の DRAM モジュールを用いると保護データと近似データの bank conflict 率は 6.25 % 程度になると予測できる。12.5 % や 6.25 % の conflict 率は無視できるとは言えないが、提案の有効性を損なうほど大きくはないため本稿では bank conflict への対処は将来の課題とする。

5. 実装

提案手法を実現するため、以下の二点を実装した。メモリアロケータは `gem5` 上に実装し、アクセサはユーザプログラムから呼び出すライブラリとして実装した。

(1) 保護データと近似データを異なる row のセット上に確保するメモリアロケータ
(2) 上記メモリアロケータを利用して保護データと近似データの同時アクセスを提供するアクセサ

5.1 メモリアロケータ

メモリアロケータは保護データを確保するための

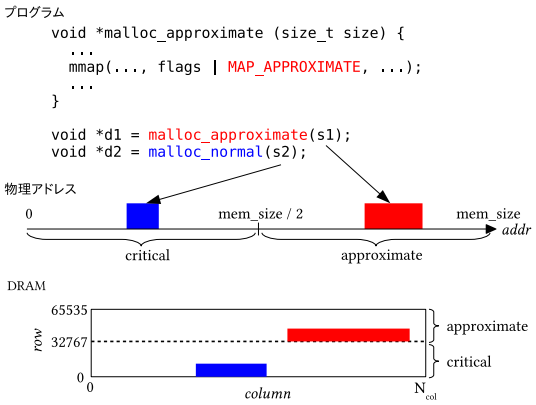

図 6 gem5 によるデータ分離の再現。malloc_approximate を用い近似データ用のメモリを確保する。本関数は内部で MAP_APPROXIMATE を指定して mmap を呼び、mmap は DRAM 上の後半の row から物理メモリを確保する。

`malloc_normal` 関数と近似データを確保するための `malloc_approximate` 関数（あわせてメモリ確保関数と呼ぶ）を提供する。それぞれの関数は初期化時に十分大きな領域を改変された `mmap` によって確保し、呼び出しがあるたびに確保した領域から要求されたサイズを切り出す。改変された `mmap` は第四引数 flags に新たなフラグ `MAP_APPROXIMATE` を指定できる。`gem5` 上の通常の `mmap` はメモリを物理アドレスの先頭から順に確保するが、改変したバージョンは `MAP_APPROXIMATE` が指定されると最大メモリサイズの半分以降の物理アドレスからメモリを確保する。これにより保護データと近似データが DRAM 上の異なる row のセットに配置される。

図 6 に、実装したメモリアロケータを用いた場合のデータ分離の例を示す。近似データ `d1` は `malloc_approximate` を用いて、保護データ `d2` は `malloc_normal` を用いて確保する。`malloc_approximate` は内部で `MAP_APPOXIMATE` フラグを指定して `mmap` を呼び出す。`MAP_APPROXIMATE` を指定すると物理アドレスの上半分から物理メモリが確保される。本稿では row を最上位に配置するメモリマッピングを想定するため、これにより row のうち半分が近似データに、残りが保護データに割り当てられる。

5.2 アクセサ

アクセサはアクセスしようとする保護データの位置、保護データの先頭位置、近似データの先頭位置を取り、指定された保護データとその保護データに対応する近似データを読み込む。アクセサの C 言語による実装を**図 7** に示す。`c_ptr`、`c_head`、`a_head` はそれぞれアクセスしようとする保護データの位置、保護データの先頭位置、近似データの先頭位置を現すポインタであり、`c_value`、`a_value` はそれぞれ読み込む保護データ、近似データを格納する変数である。C では関数の引数の型は一定であるため、近似データや保護データに任意の型を取れるよう実際には

Vol.2019-ARC-235 No.18
Vol.2019-SLDM-187 No.18
Vol.2019-EMB-50 No.18
2019/3/17

```
1 fetch(c_ptr, c_head, a_head, c_value, a_value) {
2   int index = (c_ptr) − (c_head);
3   typeof(a_head) target = (a_head) + index;
4   c_value = ∗(c_ptr);
5   a_value = ∗(target);
6 }
```

図 7 C によるアクセサの実装。c_ptr がアクセスする保護データの位置、c_head が保護データの先頭、a_head が近似データの先頭、c_value が読み込んだ保護データを保存する変数、a_value が読み込んだ近似データを保存する変数。

```
1  typedef struct {
2    int id;
3    // double score;
4  } node;
5
6  nodes = (node∗)malloc_normal(sizeof(node) ∗ N);
7  scores = (double∗)malloc_approximate(
8                                sizeof(double) ∗ N);
9
10 for(i=0; i<M; i++) {
11   int next = ...;
12   double score;
13   // id = nodes[next].id;
14   // score = nodes[next].score;
15   fetch(nodes + next, nodes, scores, id, score);
16   total_id += id;
17   total_score += score;
18 }
```

図 8 提案手法を用いたデータ分離のプログラム例

マクロで実装されている。ただしここでは可読性のため `#define` などは省略した。3 行目では `c_ptr` と `c_value` の差を `index` とする。C ではポインタ同士の加減算は型のサイズを考慮し行われるため、`index` は「アクセスしようとする保護データが先頭から何番目であるか」である。4 行目では `a_head` に `index` を足すことで保護データに対応する近似データの位置を計算する。なお `typeof(x) y` は `x` と同じ型の変数 `y` を宣言する gcc 拡張である。最後に 5 行目と 6 行目で保護データと近似データを読み込む。保護データと近似データは DRAM 上で物理的に離れた位置にあるため、DRAM 内の並列性により同時に読み込まれる。

5.3 利用例

提案手法を用いたデータ分離のプログラム例を**図 8** に示す。図はデータ構造 `node` の N 個の集合をなんらかの順序にしたがって M 回アクセスし、アクセスされた `node` の `id` と `score` の合計を計算するプログラムである。コメントアウトされた部分が Approximate Memory を想定せずデータ分離を行わないプログラム、コメントアウトされていない部分が提案手法によってデータ分離を行ったプログラムである。1 行目から 4 行目はデータ構造の定義である。データ分離を行わないプログラムでは `node` は `id` と `score` を持つが、提案手法によるプログラムでは `score` は近似データとして `node` からは削除される。6 行目と 7 行目で保護データと近似データのためのメモリを確保する。`nodes` は保護データであるため `malloc_normal` を用いて確保し、`scores` は `node` 構造体内の `score` 変数をデータ分離した近似データであるため `malloc_approximate` で確保する。この時 `nodes` と `scores` の要素数（N）は同一な必要がある。11 行目から 15 行目で `next` 番目の構造体にアクセスし `id` と `score` を読み込む。データ分離を行わないプログラムでは素直に `nodes[next]` から値を読み込む。一方、提案手法では 図 7 に示した `fetch` マクロを用い保護データと近似データを同時に読み込む。

6. 評価

6.1 評価方法

表 2 シミュレートされる環境

Core	x86_64, 8 命令同時発行 Out-of-order
Reorder Buffer	192 エントリー
L1	16 KB + 16 KB, 2 way, 32 MSHRs
L2	256 KB, 8 way, 32 MSHRs
DRAM	DDR3-1600, 1 ch, 2 ranks, 8 banks/rank

提案の有効性を評価するため、`gem5` による評価を行った。シミュレートされる環境は**表 2** の通りである。シミュレーションには Out-of-order コアのモデルとして `DerivO3CPU.py` を、DRAM のモデルとして `DRAMCtlr.py` を用い、ユーザ空間での命令のみをシミュレートしシステムコールの実行はホストへ委譲する SE モードを用いた。なお本稿はデータ分割による性能低下とその抑制が主題であり保護データと近似データが保存される row の動作速度は同一と仮定して実験を行う。評価項目は以下である。

(1) データ分離の粒度による性能低下の違い
(2) 提案手法による実行時間の変化
(3) 提案手法による row activation 数の変化

評価には以下のワークロードを用いた。各ワークロードで用いたデータサイズは 1MB である。

gemm 2 つの行列 A, B に対し、$C = A \times B$ を計算する。A, B, C 全てを疎粒度データ分離で近似データとする。

qsort 整数の列を標準 C ライブラリの `qsort` 関数でソートする。ソートされる整数列の全体を疎粒度データ分離で近似データとする。

merge sort linked list の各ノードに整数を保持し、その整数の順にリストをマージソートする。各ノード内の整数値のみを細粒度データ分離で近似データとする。

struct seq `id` と `score` を持った構造体の列をメモリに確保されたな順序で一定回数アクセスし、`id` と `score` の合計値を計算する。構造体内の `score` 値のみを細粒度データ分離で近似データとする。

struct rand `struct seq` と同様だが、データのアクセス順序がランダムである。

Vol.2019-ARC-235 No.18
Vol.2019-SLDM-187 No.18
Vol.2019-EMB-50 No.18
2019/3/17

```
1  typedef struct {
2    long id;
3    struct {
4      double score;
5      double dummy_data[D];
6    };
7  } node1;
8
9  node1 *n = (node1*)malloc_normal(sizeof(node1) * N);
10
11 // ... some init code comes here
12
13 long id_sum = 0;
14 double score_sum = 0.0;
15 for(i=0; i<N; i++) {
16   id_sum += n[i].id;
17   score_sum += n[i].score;
18 }
```

図 9 データ分離なしの場合の struct seq の構造

```
1  typedef struct {
2    long id;
3  } node2;
4
5  typedef struct {
6    double score;
7    double dummy_data[D];
8  } app_data;
9
10 node2 *n = (node2*)malloc_normal(sizeof(node2) * N);
11 app_data *a = (app_data*)malloc_approximate(
12                                 sizeof(app_data) * N);
13
14 // ... some init code comes here
15
16 long id_sum = 0;
17 double score_sum = 0.0;
18 for(i=0; i<N; i++) {
19   node2 node_loaded;
20   app_data app_data_loaded;
21   fetch(n + i, n, a, node2_loaded, app_data_loaded);
22   id_sum += node_loaded.id; // cache hit
23   score_sum += app_data_loaded.score; // cache hit
24 }
```

図 10 提案手法を適用した場合の struct seq の構造

図 9 と **図 10** に、実験で用いた `struct seq` ワークロードのソースコードを簡略化したものを示す。図 9 がデータ分離なしの場合、図 10 が提案手法を適用した場合である。データ分離なしの図の `long id` が保護データ、`double score` を含む無名構造体が近似データである。現実のアプリケーションの近似データは単一の `double` ではなくより大きなデータの場合があるため、これを模すため無名構造体には $8 \times D$ バイトの `dummy_data` を挿入する。なお C では無名構造体は存在しないかのようにアクセスでき、具体的にはソースコードの 17 行目のように内側のメンバに直接アクセス可能である。一方提案手法を適用する場合は近似データを独立して確保するために無名構造体に型 `app_data` を与えているが、C ではメモリアクセス時の型情報はサイズと先頭からの位置の決定のみに利用されるため図 9 と 図 10 で型 `app_data` の存在の有無による性能差は発生しない。

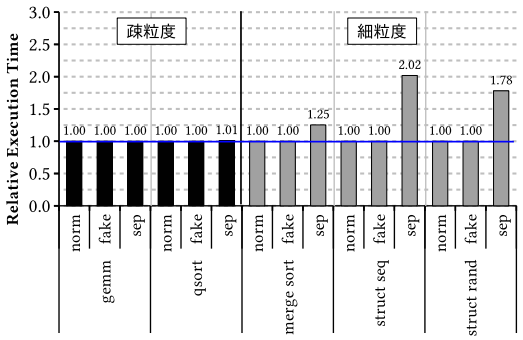

図 11 データ分離による性能低下。norm: データ分離なしの場合、fake: 利用可能な row 数を半分にするがデータ分離はしない場合、sep: 半数の row を用いてデータ分離する場合。疎粒度な分離では性能低下が見られないが、細粒度な分離では sep のみに性能低下が見られる。

6.2 性能低下のデータ分離粒度による違い

図 11 に、ナイーブなデータ分離による性能低下の評価結果を示す。ラベル `norm` はデータ分離を行わない元のワークロードの実行時間、ラベル `fake` は DRAM 上で利用できる row の数を半分に減らしたがデータ分離は行わない場合の実行時間、ラベル `sep` は DRAM 上の row 数の半分を用いてナイーブなデータ分離（ポインタを用いる）を行う場合の実行時間である。値はワークロードごとに `norm` の場合の実行時間で正規化されている。実行時間からは行列への初期値の代入などの初期化フェーズを除いた。実験結果より、次が分かる。

(1) 疎粒度データ分離では、データ分離による性能低下が見られない。これは分離されるデータの確保されるメモリ上の位置が変化するが、アクセス局所性やアクセス方法は変化しないためである。

(2) データ分離を行わず使用できる DRAM 上の row 数を減らす場合（ラベル `fake` ）は性能低下が見られない。物理アドレスからメモリデバイスへのマッピングは、割り当てる row を最上位の N ビットで決める（即ち、できるだけ少ない数の row を使う）ことが row buffer ヒット率向上の観点から一般的である。従って、データ分離を行わずに利用する row 数を減らしてもメモリデバイス上のデータ配置は変化せず、ワークロード性能も変化しない。

(3) 細粒度データ分離では、17 % から 102 % の性能低下が見られた。(2) より利用できる row 数を減らすのみでは性能は変化しないため、この性能低下は純粋にデータ分離による効果である。

シミュレーション結果より、細粒度データ分離によってワークロード性能が大きく低下することが分かった。Approximate Memory の目的は高速化と省電力であるため、

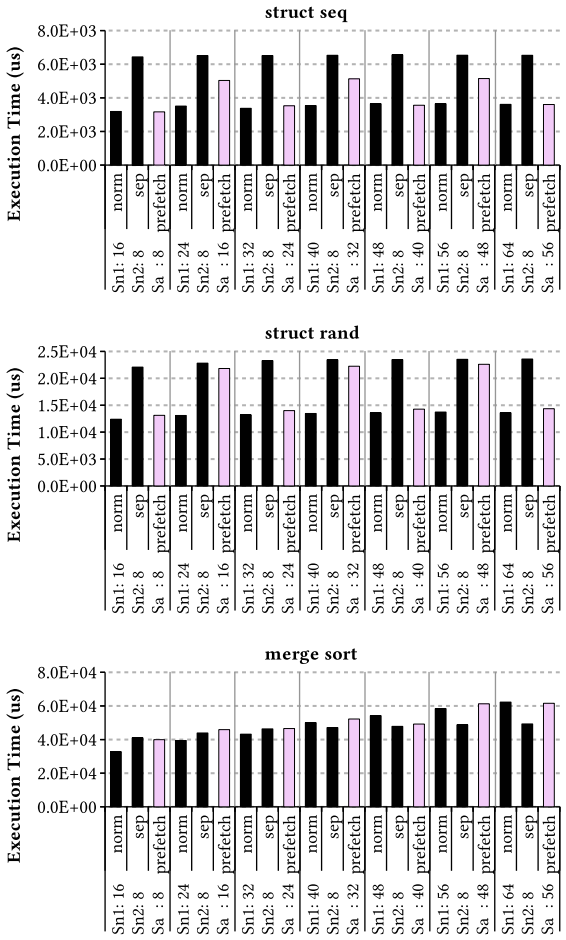

図 12 提案手法による実行時間の削減

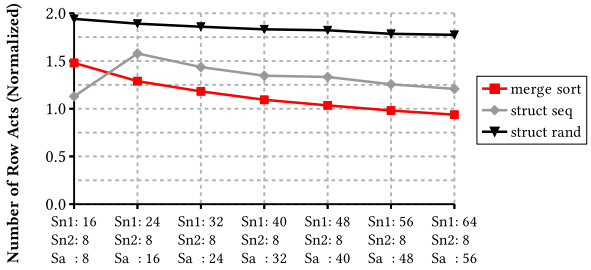

図 13 提案手法適用時の row activation 数（データ分離なしの場合の row activation 数で正規化）

データ分離によって性能が低下することは Approximate Memory の有用性を大きく損なう。従って、Approximate Memory を活用するためには提案手法によりデータ分離による性能低下の抑制が必要である。

6.3 実行時間

図 12 に各ワークロードで保護データと近似データのサイズを変化させた場合の提案手法による実行時間の変化を示す。横軸が D を変化させた場合のデータサイズ、縦軸が各ワークロードのシミュレーション環境上での実行時間である。各データサイズでの評価は 3 本の棒グラフで表され、`norm` がデータ分割なし、`sep` がナイーブなデータ分割、`prefetch` が提案手法によるデータ分割に対応する。横軸の Sn1, Sn2, Sa はそれぞれ `node1`、`node2`、`app_data` のサイズである。`node2` は `struct seq` と `struct rand` では `long id` のみを、`merge sort` では隣のノードへのポインタのみを含むため、 Sn2 は常に 8 である。また Sn1 は常に Sn2 と Sa の合計に等しい。

`struct seq` と `struct rand` は類似の傾向を示した。ナイーブな分割では全てのデータサイズにおいて実行速度が大きく低下しており、Approximate Memory を有効に活用できないことが分かる。提案手法を適用した場合の実行速度は、Sn1 が 16 の倍数のケースではデータ分割なしの場合に近い値になり、データ分割による性能低下をほぼ完全に抑制できた。一方、Sn1 が 16 の倍数ではないケースでは提案手法により性能低下がある程度抑制できたが、抑制の度合いは前述のケースに比べ小さい。データサイズの変化に応じて周期的な傾向が見られるためキャッシュライン境界や row 境界による影響であると予想されるが、具体的な原因調査は今後の課題である。

`merge sort` では Sn1 が 32 以下のケースではナイーブな分割によって実行速度が低下した。しかし提案手法を適用しても性能低下がほとんどあるいはまったく抑制できていない。また Sn1 が 40 以上のケースではナイーブな分割により実行速度が向上した。この原因については調査中だが、一つの可能性は以下である。`merge sort` では各ノードの評価値（近似データ）へのアクセスではなく次のノードを指すポインタ（保護データ）へのアクセスがクリティカルパスになり得る。データ分割しない場合には保護データと近似データがメモリ上で連続して保持されるため、 Sn1 が大きい時には次のノードを指すポインタへのアクセスでキャッシュミスが多発する。一方データ分割を行うと次のノードを指すポインタは近似データのサイズに関わらず 8 バイトおきにメモリ上に存在し常に同一の効率でアクセスできる。しかしこの論理は提案手法の場合にも同様に適用できる（提案手法では近似データのサイズに関わらず次のノードへのポインタがメモリ上連続する）ため、さらなる調査が必要である。また本結果は「細粒度データ分割により性能が低下する」とは一概に言えないことを示し、データ分割の影響を議論するためには分割の粒度のみではなくクリティカルパスの分析なども必要なことが分かった。

6.4 row activation 数

図 13 に提案手法適用による row activation 回数の変化を示す。横軸はデータサイズを表し、各値の意味は第 6.3 章と同様である。縦軸は各データサイズにおいて提案手法適用時の row activation 数をデータ分離しない場合の row activation 数で割った値である。全てのワークロード

において、近似データのサイズが大きくなるほど値が小さくっている（ただし `struct seq` の Sn1 = 16 の場合を除く）。提案手法適用下での row activation は保護データが保持される row と近似データが保持される row の二箇所で起こる。近似データのデータサイズが大きくなると近似データが保持される row の寄与が大きくなり、全てのデータが連続した row に保持されるデータ分離なしの場合に近い状態となりグラフの値が 1 に近づく。`struct seq` の Sn1 = 16 の場合に傾向が他と異なること、`merge sort` の Sn1 = 64 の場合に値が 1 より小さいことについては調査が必要である。

表 3 DRAM 消費電力のうち row activation による割合を削減するために必要な、activation 一回あたりの消費電力削減割合

	Sn1 (= sizeof(node1))						
	16	24	32	40	48	56	64
struct seq	-	.37	.30	.26	.26	.20	.17
struct rand	.48	.47	.46	.45	.45	.46	.46
merge sort	.32	.22	.15	.08	.03	-	-

表 3 に、Approximate Memory の利用によって DRAM の消費電力のうち row activation による割合を削減するために必要な row activation 一回あたりの消費電力削減割合を示す。他と傾向が異なり調査を要するケースの値は省略した。例えば `struct seq` の Sn1 = 64 のケースでは row activation 回数が 1.2 倍になっている（図 13）ため、wait time を削減するなどの既存手法により row activation 一回あたりの消費電力を $(1 - \frac{1}{1.2}) \times 100 \approx 17\%$ 削減すれば DRAM の消費電力のうち row activation が占める割合を削減できる。なお DRAM 「全体」の消費電力を議論するには本表に加え refresh や定常的な電流などの駆動時間に比例するパラメータと Approximate Memory による実行時間の削減を統合した評価が必要である。

7. 議論

7.1 データ分離の必要程度

データ分離がどの程度必要であるかは研究を要する。本稿ではデータ分離で保護データとすべきものの例としてプログラムのバイナリコードとポインタを挙げた。しかし、HPC アプリケーションではポインタにエラーが混入しても指された値を 0 とみなして計算を進めれば正しい結果に近い値が出る場合があると報告されている [11]。一方、エラー混入されたプログラムのバイナリコードを復元することはより難く、著者らの知る限り未だ実現されていない。また Approximate Memory は通常のメモリより row hammer 攻撃 [12] に弱いことから、セキュリティ上重要なデータを近似データにすることは望ましくない。従って、データ分離の必要程度について議論の余地があるが、データ分離を完全に不要にすることは難しく、本研究の重要性が低下するものではない。

7.2 データ分離の粒度以外の影響の調査

第 6.3 章の実験結果により、細粒度データ分離によってアプリケーションが低下するとは限らず逆に改善する場合があることが明らかになった。これは Approximate Memory の利用と直接関係はないが、Approximate Memory を利用する上でデータ分離は必須であるためその効果を議論する上で考慮する必要がある。今後は細粒度データ分離によって性能が改善する場合と悪化する場合の違いを定量的に明らかにし、Approximate Memory を考慮したメモリ管理に必要な要素を明らかにしていく。

7.3 プログラマインターフェースの改善

本稿ではデータ分離を実現するため、(1) 近似データ用メモリを確保するコードの実装および (2) 提案手法によるプリフェッチのためのアプリケーションコードの改変を手動で行った。しかし Approximate Memory を実際に使用する際にはこれらは自動化されることが望ましく、インターフェースの改善が必要である。例えば C++ で構造体のアクセスに用いる演算子をオーバーロードすることでユーザから見ればデータ分離なしの場合と全く同一のプログラムで提案手法を実現することは可能と予想される。

8. 関連研究

デバイスレベルで DRAM のレイテンシと消費電力を削減する手法は広く研究されている。DRAM の内部動作の待ち時間を削減しレイテンシや消費電力を削減する研究は、Approximate Memory の実現に直接応用できる。文献 [4] では activation と precharge の待ち時間削減とエラー混入率の関係を実際の DRAM で計測している。また文献 [6] や [5] では row へのアクセスの時間的局所性を用いて activation や restoration の待ち時間を削減する。これらの技術をより積極的に適用することで Approximate Memory を実現できる。一方、Approximate Memory に直接は応用できないが DRAM の内部構造を利用して効率化する研究も存在する。文献 [13] では同一のデータを複数の row に保持し見かけのキャパシタンスを上げることで activation 時間を短縮する。また文献 [14] や文献 [15] では bank 内の column を複数に分割し activation を細粒度に行うことで消費電力を削減する。これらの研究はデバイスレベルの評価に留まっており、実際にアプリケーションから利用する際のメモリ管理は考慮されていない。

文献 [16] は、Approximate Computing 用のプログラミング言語拡張 EnerJ [17] を実行するための ISA とマイクロアーキテクチャを提案する。本 ISA ではロード、ストアの命令ごとに近似のありなしを区別するため、保護データと近似データを異なるキャッシュラインに配置する必要があ

る（*"precise and approximate data never occupy the same line"*）とされている。文献 [16] ではこの保証は用意であると述べるに留まり、異なるキャッシュラインに配置することによるキャッシュミスの増加などは評価していない。文献 [8] は DRAM の refresh 頻度を下げた Approximate Memory において row をエラー耐性の高い順に並べ、保護データと近似データをそれぞれをエラー耐性の高い row 、低い row に保持する。また文献 [9] や文献 [10] でも同様に DRAM をエラー耐性ごとの bin に分割し、データの要求するエラー率ごとに分割配置する。これらの研究でも保護データと近似データの分離が必要となるが、データ分離に起因するオーバーヘッドについては議論されていない。

9. 結論と今後の課題

メモリの消費電力とレイテンシの削減に Approximate Memory が有用だが、エラー混入を許さないデータを保護するために保護データと近似データをメモリ内の離れた位置に配置するデータ分離が必須である。しかしデータ分離の粒度が小さい場合にはキャッシュミスの増加とメモリアクセス命令の増加によってアプリケーション性能が低下しうる。本稿では、分離されたデータが元々は同一の構造体内にありアクセス局所性が高いことを利用し、保護データにアクセスする際に対応する近似データをプリフェッチし性能低下を抑制する手法を提案し、シミュレーションで評価した。評価の結果提案手法はデータ分離によるアプリケーション性能低下を完全に抑制する場合がある一方、データサイズによっては性能低下を抑制できない場合があること、アプリケーションによってはデータ分離により性能が逆に向上する場合があることが明らかになった。今後はこれらのケースを詳細に分析し、Approximate Memory の利用に必要なメモリ管理方式の研究を進める。

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] Barroso, L. A., Clidaras, J. and Holzle, U.: *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, Morgan & Claypool Publishers (2013).

[2] Meisner, D., Gold, B. T. and Wenisch, T. F.: PowerNap: Eliminating Server Idle Power, *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 205–216 (2009).

[3] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006).

[4] 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).

[5] 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).

[6] 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).

[7] Jacob, B., Ng, S. and Wang, D.: *Memory Systems: Cache, DRAM, Disk*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007).

[8] 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).

[9] 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).

[10] Venkatesan, R. K., Herr, S. and Rotenberg, E.: Retention-aware placement in DRAM (RAPID): software methods for quasi-non-volatile DRAM, *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 155–165 (2006).

[11] Fang, B., Guan, Q., Debardeleben, N., Pattabiraman, K. and Ripeanu, M.: LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures, *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pp. 117–130 (2017).

[12] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K. and Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors, *International Symposium on Computer Architecture (ISCA)*, pp. 361–372 (2014).

[13] Choi, J., Shin, W., Jang, J., Suh, J., Kwon, Y., Moon, Y. and Kim, L.: Multiple Clone Row DRAM: A low latency and area optimized DRAM, *International Symposium on Computer Architecture (ISCA)*, pp. 223–234 (2015).

[14] Lee, Y., Kim, H., Hong, S. and Kim, S.: Partial Row Activation for Low-Power DRAM System, *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 217–228 (2017).

[15] Zhang, T., Chen, K., Xu, C., Sun, G., Wang, T. and Xie, Y.: Half-DRAM: A high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation, *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 349–360 (2014).

[16] Esmaeilzadeh, H., Sampson, A., Ceze, L. and Burger, D.: Architecture Support for Disciplined Approximate Programming, *SIGARCH Comput. Archit. News*, Vol. 40, No. 1, pp. 301–312 (2012).

[17] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L. and Grossman, D.: EnerJ: Approximate Data Types for Safe and General Low-power Computation, *Programming Language Design and Implementation (PLDI)*, pp. 164–174 (2011).