Vol.2017-OS-141 No.12
2017/7/27

# 性能カウンタを用いた近似実行の高速なエミュレーション

穐山 空道[1,a)]　広渕 崇宏[1]

**概要**：データセンタ省電力化のため計算精度を落とす代わりに消費電力を削減する近似実行（Approximate Computing）が着目され、特に DRAM モジュールに適用し大容量化・消費電力増が進むメモリサブシステムを省電力化する研究が盛んである。近似実行では大幅な電力削減が可能な一方、データ化けによる計算誤差やクラッシュ等アプリケーションへの影響も大きく、実際のアプリケーションに対し近似実行の適用可能性の調査が必須である。適用可能性の調査には、どのデータを近似するか、DRAM の電力をどの程度削減するか等様々なパラメータが存在し、従って近似実行を活用するためにアプリケーションへの影響を様々なパラメータで軽量に調査できることが重要である。しかし既存研究ではメモリトレースツールやハードウェアエミュレータが利用され、これらは実機の数百倍から千倍程度低速である。そこで本研究では、コモデティな CPU のハードウェア機能を利用することでメモリへの近似実行がアプリケーションに与える影響を高速に見積もる手法を提案する。近似を許すデータと許さないデータを別々の NUMA ノードに配置しメモリコントローラーの性能カウンタから各データへの IO 量を取得することで、近似データへのエラー混入が計算結果に与える影響をハードウェアシミュレータなしに再現する。評価の結果、提案機構ではアプリケーションへの影響を実機の数倍程度の時間で見積もれることを確認した。

## 1. 序論

データセンタの消費電力を削減することは重要な課題であり、特にマシン消費電力のうちメモリサブシステムの割合が増大していることが報告されている。これはビッグデータ分析や人工知能アプリケーションなど巨大なデータを処理するワークロードのために、搭載メモリが大容量化しているためである。メモリサブシステムの消費電力増大に対処するため、従来型の DRAM 消費電力を下げる研究 [12,17] や、不揮発性メモリに代表される新たなデバイスを利用する研究 [9,10] などが盛んである。

計算機の消費電力を削減する革新的な手法として、計算精度を落とすことで消費電力を削減する手法である Approximate Computing（以下近似実行と呼ぶ）が注目されている。近似実行は計算結果の一意性を要求しないことで、正確な計算結果を要求する場合では実現不可能なレベルでの消費電力削減を可能にする。前述のようにメモリサブシステムの消費電力はマシン内の多くを占めるため、近似実行をメモリに適用する研究が盛んである [6,13,16]。

近似実行では正確な計算が保証されないため、対象とするアプリケーションに応じ消費電力削減効果とアプリケーションへの影響の関係評価が必須である。アプリケーションへの影響には計算結果に数値的誤差が混入するといった許容可能なものや、アプリケーションが正常に終了しないなどの許容できないもの等がある。また誤差の大きさや正常終了しない確率は一般に消費電力削減効果が大きいほど高くなる。影響の評価にはアプリケーション中のエラーの混入を許すデータ（以下「近似データ」と呼ぶ）とエラーの混入を許さないデータ（以下「厳密データ」と呼ぶ）の割合（どのデータを近似データとしどのデータを厳密データとするか）や近似実行システムの消費電力に関する設定など多くのパラメータが存在し、近似実行の特定のアプリケーションへの適用可能性を評価するためにはパラメータの変更による影響を繰り返し評価する必要がある。

メモリに対する近似実行のモデルには、DRAM のリフレッシュレートを下げるものや DRAM セルの電荷のチャージを通常短時間に行うものが存在し、いずれにおいてもアプリケーションへの影響評価には CPU から各 DRAM セルへのアクセス回数が必要である。例えば DRAM のリフレッシュレートを下げて消費電力を削減するモデルでは、リフレッシュが行われないことによる電荷の消失よってデータにエラーが発生する。しかし CPU コアからあるセルへアクセスがあると当該セルには電荷がチャージされるため混入されたエラーの影響は低減する。

既存研究ではプログラムのメモリトレースとハードウェ

[1] 国立研究開発法人 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology (AIST)
[a)] s.akiyama@aist.go.jp

アのシミュレータを用い CPU コアから DRAM セルへのアクセス回数を見積もることで近似実行のアプリケーションへの影響を評価する。しかしこれらは実機の数百倍から千倍程度低速であり、近似・厳密データの別やエラー率等のパラメータを何度も変更しアプリケーションへの影響を評価できない。そこで本稿では、CPU コアから DRAM セルへのアクセス回数を実機を利用して見積もり、メモリへの近似実行のアプリケーションへの影響の軽量な評価手法を提案する。近年の CPU には様々なパフォーマンスメトリックを計測するハードウェアカウンタが備わっており、本稿ではメモリコントローラに搭載された IO 量を計測するカウンタを利用する。近似データと厳密データを異なる NUMA ノードに配置しそれぞれのデータへの IO 量を独立に計測することで、CPU キャッシュの影響を考慮して近似データへのアクセス回数を計測できる。また将来的には CPU コアに搭載されたアクセス先アドレスを記録するカウンタを用いてより細粒度なアクセス回数見積もりを行う。提案法により SPEC CPU 2006 ベンチマークへの近似実行の影響を調査した結果、実機での実行とほぼ同等から 5 倍程度の実行時間で結果が得られることが分かった。これは同一ベンチマークのメモリトレースを取得するのにかかる時間と比較して 480 倍の改善である。

本論文の構成は以下である。第 2 章で既存研究とその課題を示す。第 3 章で本稿で仮定する近似実行のモデルとその先行研究を示す。第 4 章で提案手法とその実装について説明する。第 5 章では提案手法による近似実行の速度および得られたアプリケーションへの影響を議論する。第 6 章では現状の課題点についての議論と将来の展望を述べ、第 7 章で本稿を結論する。

## 2. 関連研究とその課題

### 2.1 メモリへの近似実行の適用

近似実行による消費電力の削減は広く研究されている。文献 [13] では DRAM のリフレッシュ頻度をデータの種類によって調整することでメモリサブシステムの消費電力を削減する。DRAM のセルを構成するキャパシタは非常に微細であるため、1 秒に何度もリフレッシュしなければデータが消失する。この文献ではプログラマがエラー混入を許すデータを指定し、システムが当該データを通常のデータとは異なるアドレス空間に配置する。近年の DRAM には Partial Array Self Refresh (PARS) [5] が実装されており、Bank 毎にリフレッシュ頻度を指定できる。この文献ではモバイルデバイスにおいて DRAM の消費電力の 20%から 25%（デバイス全体の電力の 1%相当）の削減を無視できる程度の計算結果への影響で達成した。

文献 [16] [6] では、近似データに混入されたエラーが厳密データに伝播しないことを言語レベルで保証する。文献 [13] 等の既存研究ではエラー混入を許すと指定されたデータは低消費電力で保持されるが、計算途中でエラー混入を許さないデータにエラーが伝播しないと保証することはプログラマの責任である。一方、この文献ではプログラミング言語を拡張し静的解析を行うことでこれを自動的に保証する。文献 [13] と同様に DRAM のリフレッシュ頻度を下げるのみならず、CPU 内のレジスタや計算ユニットの消費電力も低減することで低消費電力なシステムを実現する。

### 2.2 既存研究におけるアプリケーションへの影響評価

近似実行の有効性評価のため、アプリケーション結果への影響の定量的評価が必須である。文献 [13] では、アプリケーションへの影響を見積もるために通常より低いリフレッシュ頻度によるエラーの混入をエミュレートする。リフレッシュ頻度が低い bank に属していても頻繁にアクセスされるメモリセルはアクセス時に発生するデータ書き戻しにより電荷が保たれる。メモリセルへのアクセスの有無は CPU のキャッシュの影響によってソフトウェアから見たアクセス履歴とは異なるため、この影響を考慮するためにこの文献では PIN [14] によるメモリトレースに加えハードウェアのシミュレータを用いる。文献 [16] でも同様にアプリケーションへの影響を評価するためにメモリアクセスや演算などをトレースしエラーの混入をエミュレートする。

これらの文献ではエミュレーションにかかる時間は報告されていないが、一般にサイクルレベルでのハードウェアエミュレーションは実機の数百倍から千倍程度低速である。またメモリアクセストレースの取得も非常に低速であり、実際に文献 [4] では SPEC CPU 2006 ベンチマークからの PIN を用いたメモリトレース取得に 1 プログラムあたり 4 日から 6 日かかるとしている。アプリケーションへの影響の大きさを決める要因は発生させるエラー率や近似・厳密データの選択など様々なパラメータがあり、設計者や利用者が最適なパラメータを発見する必要がある。従ってハードウェアエミュレータを用いる手法ではアプリケーションへの影響を効率良く評価できない。

また文献 [2] ではプログラムのデータフローを解析しエラーの伝播を検知する。近似実行ハードウェアのエラー発生確率からプログラムを当該ハードウェア上で実行した時の出力の信頼度を確率的に見積もることが可能である。しかし本手法ではプログラムを静的に解析するが実際に近似実行を行うわけではないため、見積もりが “conservative” だと報告されている。具体的には、ループ内に前回のループに依存する値がある場合、その値の信頼度はループをあり得る最大まで回った場合の値（最大回数が事前に決定可能な場合）またはゼロ（ループの最大回数が決定不能な場合）となる。従って、近似実行のアプリケーションへの影響を精度よく見積もるには実際に対象アプリケーションを実行することが必要である。

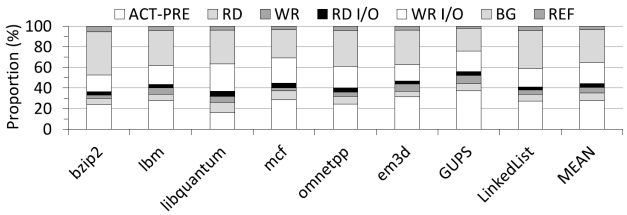


**図 1** 文献 [12] の Figure 2 より引用。ワークロード実行中の DRAM 消費電力の内訳。各ラベルはそれぞれ DRAM のオペレーションにかかる電力の割合（詳細は文献 [12] を参照）。本稿ではデータ読み書きにかかる電荷の移動である ACT と、読み書き後の電荷の再チャージである PRE を削減することを目指す。

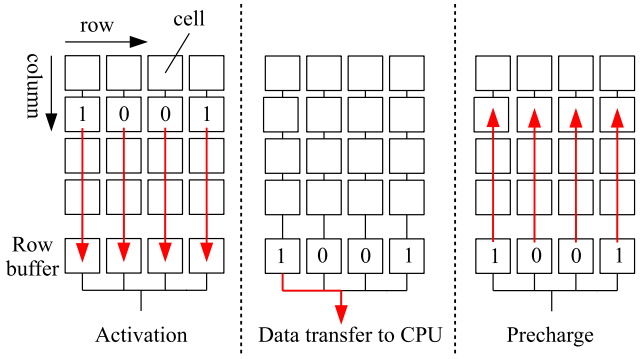


**図 2** DRAM の内部構造とデータ読み出し

## 3. メモリへの近似実行の適用

### 3.1 DRAM の消費電力内訳

本稿では DRAM の消費電力のうち、データの読み書きにかかる部分を削減するモデルを用いる。文献 [13] などの既存研究では電荷のリフレッシュにかかる電力を削減する点が異なる。提案するエミュレーション機構の電荷リフレッシュにかかる電力を削減する研究への適用については 6.3 節で述べる。

**図 1**（文献 [12] の Figure 2 より引用）は、様々なアプリケーションを実行中の DRAM モジュールの消費電力の内訳を表す。横軸はワークロードを表し、積み上げグラフの各ラベルは DRAM の内部動作を表す。ACT-PRE はアクチベーションとプリチャージの合計で、それぞれデータ読み書き時のバッファへの電荷の移動および読み書き終了後のメモリセルへの電荷の再チャージを表す。また REF は電荷のリフレッシュにかかる電力である。ワークロードの詳細および他のラベルについては文献 [12] を参照のこと。

図 1 より、ACT-PRE にかかる電力は多くのワークロードで DRAM の消費電力の最大あるいは 2 番目に大きな部分を占める。そこで本稿では、ACT-PRE で行う電荷のチャージを通常より短時間に行うことで、近似データへのエラーの混入を許す代わりに低消費電力な実行を目指す。3.2 節で DRAM のデータ読み書きの内部動作について説明し、 3.3 節で電荷の短時間チャージによる消費電力削減の原理について記し 3.4 節に本稿で仮定するエラー混入のモデルを導入する。

### 3.2 DRAM の内部構造とデータの読み書き

**図 2** に DRAM モジュールの内部構造およびデータ読み出し動作の概略を示す。実際の DRAM モジュールは帯域向上のため複数の類似の構造が並列にアクセスされる点で異なるが、本節で説明される内容については同様の議論が成り立つ。DRAM モジュールは小さなキャパシタ（コンデンサ）の集合であり、1 つのキャパシタが 1 ビットのデータを保持する。キャパシタは row と column に並んでおり、512 row と 512 column の集合を MAT と呼ぶ。1 つの MAT から 1 ビットが転送され、512 個の MAT に並列にアクセスすることで 64 バイトの転送をセルへのアクセス 1 回と同等の遅延で可能にする。またある row のデータを一時的に保持する row buffer が備わり、CPU へのデータの転送は row buffer から行われる。

DRAM からのデータ読み出しには、対象ビットの属する row から row buffer への電荷の移動（アクティベーション）と row buffer から元の row への電荷の書き戻し（プリチャージ）が発生する。アクティベーションでビットを保持した電荷が row buffer に保持されることで、同一 row 内のデータに連続してアクセスした際に高速アクセスが可能である。これを row buffer hit と呼ぶ。またプリチャージによって読み出された電荷が row buffer から元の row に書き戻される。これを row を close すると言う。

Row buffer がヒットすればアクティベーション、プリチャージにかかる電力およびレイテンシが削減されるため、row buffer ヒット率を向上させる研究は盛んである [15,18]。しかし、CPU コアの増大にともない複数のスレッドが異なるアクセスパターンでメモリアクセスを行うため、一般に row buffer のヒット率は特に書き込みにおいて低いと報告されている [12]。従って row のアクティベーション、プリチャージにかかる電力は未解決の問題である。

### 3.3 電荷の短時間チャージによる消費電力削減

本稿では、アクチベーションとプリチャージでの電荷チャージを通常よりも短時間に行うことで、近似データへのエラーの混入を許す代わりにメモリサブシステムの消費電力を削減するモデルを提案する。電荷のチャージにかかる電流を流す時間が削減されることで消費電力を削減する。3.1 節の通りアクチベーションとプリチャージは DRAM の消費電力の大きな部分を占めるため、本モデルにより大きな電力削減が期待できる。

アクチベーションとプリチャージでの電荷の短時間チャージは Chang らにより議論されている [3]。この文献ではメモリアクセスの遅延低減を目的として DRAM モジュール

の仕様で定められたチャージ完了にかかる待ち時間を意図的に短縮する。例えば、DRAM モジュールの仕様によりアクチベーション（読み出すデータの所属する row から row buffer へのデータの移動）の完了待ち時間は 13.125 ns と定められているが、これを 10 ns まで削減してもビットエラー率が $10^{-10}$ 未満に抑えられると報告されている。また 7.5 ns まで削減すると平均ビットエラー率は $10^{-2}$ から $10^{-3}$ 程度になると報告されている。

提案手法ではアクチベーションとプリチャージでの電荷の短時間チャージによって DRAM の消費電力削減を目指すが、Chang らや類似研究による消費電力削減効果は我々の知る限り議論されていない。そこで本稿では、「DRAM のある操作にかかる消費電力はその待機時間に比例する」と仮定する。具体的には、アクチベーションを 13.125 ns から 10 ns に高速化した場合、電荷移動にかかる電力のうち $1 - 10/13.125 \approx 13.8\%$ が削減されるとする。 図 1 から例えば `bzip2` ベンチマークではアクチベーションとプリチャージの合計消費電力は DRAM 全体の消費電力の 20%強であり、これを 13.8% 削減した場 opo 合の消費電力削減は DRAM 全体の約 2.7%となる。

### 3.4 本稿でのエラーモデル

本稿ではアクチベーションとプリチャージでの電荷チャージを通常より短時間で行うことで、メモリアクセス 1 回あたり一定の確率でビットエラーが発生すると仮定する。すなわち、あるビットへの読み込みまたは書き込みが $N$ 回発生し、一回あたりのビットエラー率が $R$ であるとすると、当該ビットにエラーが発生する（ビットが反転する）確率は $1-(1-R)^N$ である[*1]。展開すると $NR - N(N-1)/2 \times R^2 + ...$ であるが、$NR$ が 1 より十分小さい場合には $R$ が 2 次以上の項を 0 に近似でき結局 $NR$ となる。つまり例えば $R = 10^{-10}$ の時、あるビットに 10 回アクセスした場合にビット反転する確率はほぼ $10^{-9}$ となる。本モデルと Chang らの研究 [3] で調査されている実際の DRAM モジュールのエラー発生方式の違い及びその違いを考慮した今後の展開については 6.1 節に記す。

本稿のエラーモデルでは発生するエラーをエミュレートするにはあるビットにアクセスされた回数が必要である。CPU キャッシュの影響により、これはソフトウェアがあるアドレスを含む範囲に load/store 命令を発行した回数とは大きく異なる。従って既存の方式での本エラーモデルのアプリケーションへの影響調査には PIN 等によるメモリトレースをキャッシュエミュレータに入力する、またはプログラムのハードウェアエミュレータ上での実行が必要であり、エラー率の変化や近似データの選び方などのパラメータを効率よく探索できない。次章以降でこの問題を解決するための機構について提案と評価を行う。

[*1] 「N 回続けてエラーが起こらない確率」を 1 から引く

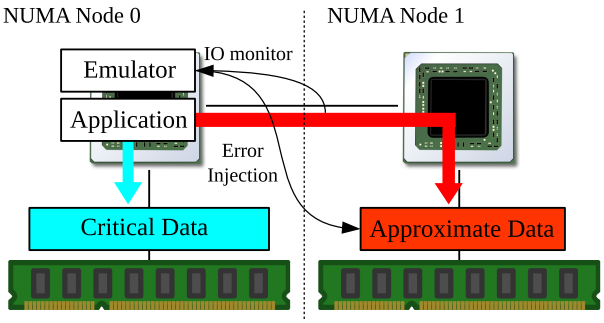


**図 3** 提案システムの概略。厳密データと近似データを異なる NUMA ノードに配置し、近似データの存在する NUMA ノードの IO 量を計測することでアクセス回数を見積もる。見積もられたアクセス回数に応じて近似データにエラーを混入する。

## 4. 近似実行の高速なエミュレーション

### 4.1 提案システム

本稿では、メモリサブシステムに対し近似実行を適用した際にアプリケーションに与える影響を高速に評価するシステムを提案する。提案システムは近似データと厳密データを分離しそれぞれへの IO 量を独立に計測することで、ハードウェアエミュレータを用いずに近似データのみへのアクセス回数を取得できる。提案システムの概要を **図 3** に示す。提案システムの動作は以下のようになる。

(1) 厳密データと近似データを異なる NUMA ノードに配置し、各データへのアクセスを分離する。
(2) メモリコントローラの性能カウンタから各 NUMA ノードへの IO 量を計測する。CPU コアから DRAM 上の近似データへ実際のキャッシュの影響を考慮したアクセス回数が見積もれる。
(3) CPU コアの性能カウンタからキャッシュをミスした load 命令の対象アドレスを取得し、近似データ内のキャッシュライン毎のアクセス回数を見積もる。
(4) 計測された近似データへのアクセス回数を基に、近似データに人為的なエラーを混入する。

以下では近似データの分離、IO 量の計測、エラーの混入についてについて実装の詳細を記述する。(3) についてはアクセス回数の見積りを細粒度に行うために今後実施する予定であり、6.1 節に詳細を記す。

### 4.2 厳密データと近似データの分離

近似データと厳密データの分離は、それぞれを異なる NUMA ノードに配置し実現する。すなわち提案システムの動作には NUMA ノードが 2 個以上必要である。各 NUMA ノード（CPU ソケット）は独立したメモリコントローラを持ち、CPU コアから近似データへの IO 量のみを独立に計測できる。

本稿では Linux の NUMA API を用い、`malloc` に類似

したインターフェースである bmalloc を実装した。具体的には、近似データ以外の全てのデータ（対象アプリケーションのコード、スタック、ヒープ）を NUMA API によって NUMA ノード **0** に配置し対象アプリケーションを起動する。その後アプリケーションが近似データを bmalloc で確保した領域に格納することで、近似データは NUMA ノード **1** に配置される。NUMA ノード 1 のメモリコントローラから IO 量を取得し、実際に近似データの保持された DRAM セルへのアクセス回数を CPU キャッシュの影響を考慮して算出できる。

近似データの分離にはアプリケーションが明示的に bmalloc を使う必要がありプログラマの補助を要するが、本仮定は以下の 2 点により妥当と言える。

(1) 類似の仮定は既存研究においても広く行われている。例えば [16] ではプログラマが型名にアノテーションを付与することで近似データを区別する。
(2) どのデータを近似するかの選択も近似実行の対象アプリケーションに対するパラメータの一つであり、ユーザが様々に変更して影響を調査することが必要である。

以上から本稿でも近似データと厳密データの区別はプログラマが明示的に与えるものとする。

### 4.3 エラーの混入

近似データへのエラーの混入はエミュレータの IO 量監視スレッドが行う。当該スレッドはメモリコントローラ内の性能カウンタを定期的に読み出すことで NUMA ノード 1 の IO 量（すなわち近似データへの IO 量）を監視する。近似データへの IO 量が一定値に達すると IO 量監視スレッドはシグナルを発生させ、bmalloc が動作するスレッドに IO 量が閾値を越えたことを通知する。bmalloc はアプリケーションにリンクして利用するため、このシグナルによってアプリケーションは動作を一旦停止する。bmalloc は管理するメモリ領域（アプリケーションが bmalloc で確保した領域）に対し指定されたエラー率でエラーを混入し、シグナルハンドラを終了してアプリケーションに処理を戻す。

エラーの混入は、メモリ領域に格納されたデータの型により異なる処理を行う。格納されたデータが整数型である場合にはエラー率に従って単純にビットを反転する。一方、格納されたデータの型が浮動小数点数の場合、ランダムなビットを反転すると Not a Number となりアプリケーションが強制終了する場合がある。そこで提案システムでは bmalloc に通常の malloc から追加の情報として格納するデータの型をプログラマが与える。データが浮動小数点数である場合、エラーの混入はビットの反転ではなくランダムな値の代入によって行う。

あるビットが反転する確率の計算には、1 アクセスあたりのエラー率に加えて当該ビットに何回アクセスされたかたが必要である（3.4 節）。本稿では簡単のため、計測された IO が近似データの全ビットに均等に発生したと仮定する。例えばエラー混入を行う IO 量の閾値が 10 GB、近似データの量が 200 MB、エラー率が 1 アクセスあたり $10^{-10}$ の時、IO 量が閾値に達した際の各ビットへのアクセスは $10,000/(200*8) = 6.25$ 回、各ビットのエラー発生率は 3.4 節に示した近似を用いるとおよそ $6.25 \times 10^{-10}$ となる。将来的には各ビットへの正確なアクセス回数をハードウェアエミュレータなしに見積もる予定であり、Intel PEBS を用いた将来の方針を 6.1 節に示す。

**表 1** 評価環境

| | |
|---|---|
| OS | Debian GNU/Linux 8 (Linux kernel 4.9) |
| CPU | Intel Xeon E5-2630 v4 ×2 |
| LLC Size | 25 MB (per NUMA node) |
| Memory | 64 GB (per NUMA node) |

## 5. 評価

### 5.1 評価環境と方法

**表 1** に評価環境を示す。カーネルは Debian GNU/Linux のデフォルト版（3.16）から最近のもの（4.9）に更新されている。これは Linux 3.16 にはパフォーマンスカウンタ関連のバグがあり、今後用いる予定の PEBS を正しく扱えないためである。LLC Size、Memory は NUMA ノードごと（ソケットごと）の値であり、マシン全体では 2 台の CPU を搭載するため 2 倍の値となる。

評価は SPEC CPU 2006 ベンチマークに含まれる以下のワークロードについて行った。

**milc** 量子力学における粒子間相互作用のシミュレーションを行う。ベクトルや行列の演算が主である。
**mcf** 公共交通機関の目標タイムテーブルから最適な車両スケジューリングを決定する。シンプレックス法による最小流問題に帰着して計算される。

それぞれのアプリケーションについてソースコードを変更し近似データを分離し、近似実行のエミュレーションにかかる時間、近似実行によるアプリケーションへの影響、近似データ分離の影響による IO 量の変化について評価した。

### 5.2 ソースコード変更による近似データ分離

**図 4** および **図 5** に、近似データを分離するための milc および mcf のソースコードの変更点を示す。なお SPEC CPU benchmark はフリーソフトウェアでないため、図には実際のソースコードではなく変更部分を概念的に示した。

milc では double 型の組で表されるベクトルや行列が、元のソースコードで直接 malloc される。本稿ではこのうちベクトルを確保する部分を bmalloc に直接置き換えることで近似データを分離する。データへのアクセスは元々ポインタ経由で行われるため変更は必要ない。また mcf では

```
1  ...
2  // double *vec = malloc(...); // original
3  double *vec = bmalloc(...); // modified
4  ...
```

**図 4** milc における近似データ分離のためのコード変更

```
1  struct X {
2    // int value; // original
3    int *value; // modified
4    struct X *neighbor;
5    ...
6  };
7
8  struct X *data = malloc(sizeof(struct X) * n);
9
10 /** newly inserted code starts **/
11 int *values = bmalloc(sizeof(int) * n);
12 for(i=0; i<n; i++){
13   data[i].value = values[i];
14 }
15 /** newly inserted code ends **/
16
17 // data[i].value = 123; // original
18 *(data[i].value) = 123; // modified
19 ...
```

**図 5** mcf における近似データ分離のためのコード変更

最小流問題のグラフ構造が構造体で表され、構造体は近似してよい値の他に隣接ノードへのポインタ等を含む。本稿では分離のために近似してよいデータを格納するメモリ領域を `bmalloc` で確保し、元の構造体に確保された領域へのポインタを代入する。また構造体のメンバへのアクセスを直接アクセスから pointer defererence に変更する。

近似データの分離はメモリ上でのデータレイアウトの変更を伴い、アクセスの局所性が変更前のプログラムから変化する。従ってキャッシュのヒット率なども変化する可能性があり、近似データへのキャッシュの影響を考慮したアクセス量の見積もりに影響を与え得る。これについて 5.5 節で評価する。

### 5.3 エミュレーション速度

提案手法による近似実行エミュレーション速度を確認するため、無改変の SPEC CPU 2006 ベンチマークと提案手法を適用した場合の SPEC CPU 2006 ベンチマークの実行速度を比較した。入力データのサイズは 2 番目に大きい train を使用した。エラーの混入はエラー率を 1 アクセスあたり $10^{-11}$ に設定し、IO 量の閾値を `mcf` では 10 GB、`milc` では 300 MB に設定した。この閾値はエラーの混入が数秒に一度起こるような値を選んだ。

**図 6** に実験結果を示す。各ベンチマークのバーは左から

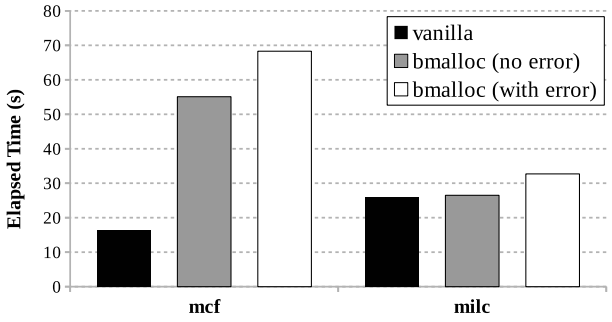


**図 6** 提案手法を適用した場合のアプリケーション実行時間。vanilla：無改変の状態。bmalloc (no error)：近似データを分離するがエラー混入はしない場合。bmalloc (with error)：近似データへのエラー混入を行った場合。

ら無改変な状態での実行時間（vanilla）、bmalloc により近似データの分離のみ行った場合の実行時間（bmalloc no error）、bmalloc により近似データの分離とエラーの混入を行った場合の実行時間（bmalloc with error）である。vanila と bmalloc no error の差分が近似データの分離によるオーバーヘッド、bmalloc with error と bmalloc no error の差分がエラー混入処理によるオーバーヘッドである。

`milc` ベンチマークでは近似データの分離によるオーバーヘッド、エラー混入処理によるオーバーヘッド共に無視できる大きさである。一方 `mcf` ベンチマークでは近似データの分離により実行速度が 3.4 倍に低下し、エラー混入処理によって 4.2 倍に低下する。対象のアプリケーションによってオーバーヘッド量が大きく異なる原因は近似データの分離方式の違いであり、これについて 5.5 節で分析を行う。

本実験では最大 4.2 倍性能が低下したが、これは既存研究からは大きな改善である。既存研究では近似実行によるアプリケーションへの影響をハードウェアシミュレータやメモリトレース取得ツールを用い評価する。これらの上でのアプリケーション実行は実機の数百倍から千倍程度低速である。例えば、SPEC CPU 2006 ベンチマークからの Intel PIN [14] によるメモリトレース取得は 1 プログラムあたり 4 日から 6 日かかる [4]。 表 1 の環境で mcf を ref データセット（最大サイズのデータセット）で実行すると 260 秒かかり、これが PIN 上では 4 日（$60 \times 60 \times 24 \times 4 = 345600$ 秒）かかると仮定すると、データセットが train の場合（図 6 では 16.4 秒）のメモリトレース取得にかかる時間は $345600 \times 16.4 \div 260 \approx 21800$ 秒と見積もれる。提案手法での近似実行にかかる実行時間は図 6 から約 68 秒であり、これは既存手法より約 480 倍高速である。

### 5.4 ケーススタディ：アプリケーションへの影響調査

提案機構を用いて、本稿で提案する近似実行モデルで 1 回のメモリアクセスあたりのエラー率を変化させた場合の SPEC CPU 2006 ベンチマークへの影響を調査する。エラーを混入する IO 量の閾値は `mcf` では 10 GB、`milc` で

**表 2** 近似実行の mcf ベンチマークへの影響

| | $R$ = 1e-11 | $R$ = 5e-11 | $R$ = 1e-10 |
|---|---|---|---|
| 成功率 | 70 % | 40 % | 10 % |
| エラー混入数 | 6 bits | 21 bits | 46 bits |
| 実行可能率 | 100 % | 90 % | 80 % |

**表 3** 近似実行の milc ベンチマークへの影響

| | $R$ = 1e-11 | $R$ = 1e-10 | $R$ = 1e-9 |
|---|---|---|---|
| 平均計算誤差 | 4.5e-8 | 8.1e-7 | 4.2e-6 |
| エラー混入数 | 20 bits | 199 bits | 2031 bits |
| 実行可能率 | 100 % | 100 % | 100 % |

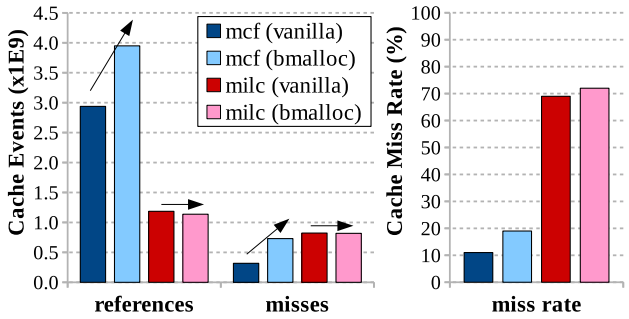


**図 7** 各アプリケーションを改変しない場合（vanilla）と近似データを分離した場合（bmalloc）のキャッシュ参照数およびキャッシュミス回数（左図）とキャッシュミス率（右図）

は 300 MB に設定した。この閾値はエラー混入が数秒に一度起こるような値を選んだ。

各ベンチマークについて、エラー率を変化させた場合の実行結果への影響、エラー混入数、実行可能率を評価した。実行結果への影響は各アプリケーションによって異なる。`mcf` はスケジューリング問題を解くベンチマークであり探索結果が出力される。ここでは近似実行で出力されるスケジュールが通常実行で出力されるものと同一のとき成功、そうでないとき失敗と定義する。`milc` では出力結果のうち GACTION（ゲージ作用の強さ）の数値を近似実行と通常実行で比較する。またエラー混入数は 1 回の実行中に混入されたビットエラーの数の平均、実行可能率はアプリケーションが正常に終了した割合である。例えば `mcf` のような探索問題ではエラーを多く混入すると探索が収束せずプログラムが終了しないことが考えられる。

**表 2** に `mcf` での評価結果を示す。表の $R$ は 1 アクセスあたりのエラー率を表す。エラー率 1e-11 では 70%の成功率で実行可能率は 100%だった一方、エラー率 1e-10 では成功率 10%、実行可能率 80%に低下した。表から成功率はエラー率に対してセンシティブであり、本研究のようにパラメータを変更して近似実行の影響を軽量に調査できることが重要である。

**表 3** に `milc` での評価結果を示す。表の $R$ は 1 アクセスあたりのエラー率を表し、 表 2 とパラメータが異なることに注意が必要である。`milc` では近似データが `double` 型であり、NaN の発生を防ぐためエラーの混入はビット反転ではなくランダムな値を設定した。すなわちエラー混入数が $N$ bits とは `double` 型近似データ $N$ 個をランダムな値に設定したことを意味する。なおエラーを混入しない通常実行での GACTION の値は 2.222088 である。本ベンチマークは物理的状態の遷移をシミュレーションするため、実行可能率は常に 100%であった。表からエラー率を 1e-11 から 1e-9 と 100 倍にすると GACTION の誤差も 100 倍になり、GACTION の計算誤差はエラー率に対しセンシティブであると分かる。

本節のケーススタディにより以下の 2 点が示された。

(1) 提案手法により実際に近似実行のパラメータを変化させアプリケーションへの影響を軽量に調査できる。
(2) アプリケーションの結果はエラー率の変化にセンシティブであり、また各アプリケーションによりその度合いは異なる。

またエラー率に以外にも、例えば `milc` では今回近似データとしたベクトルの他に行列データを近似することが考えらる。近似実行のアプリケーションへの影響を調査するには様々なパラメータを変更して何度も実行する必要がある。提案手法は実機に近い実行時間でアプリケーションに与える影響を調査でき、この問題に対して極めて有用である。

### 5.5 データ分離による IO 量の変化

提案手法で実行されるプログラムは、メモリ上のデータレイアウトが元のプログラムとは異なる。これは数値的な計算結果には影響を与えないが、データアクセスの局所性に影響を与えるためキャッシュの影響を考慮したメモリアクセス回数を正確に見積もれない可能性がある。そこで本節では、ケーススタディで使用したベンチマークについて、通常実行時と提案手法で近似データを分離した場合のキャッシュミス数を比較する。なお提案手法では近似データの分離のみ行い、エラーの混入は行わない。また近似データの分離方法は 5.2 節に示したものと同様である。

**図 7** は各アプリケーションで近似データを分離する場合としない場合のデータアクセスパターンの変化を示す。vanilla はアプリケーションを改変しない場合、bmalloc は近似データを分離した場合である。図の左側がキャッシュ参照数およびキャッシュミス数を、右側がキャッシュミス率を示す。なおキャッシュ参照数およびミス数の $y$ 軸は実際の値を $10^9$ で割ったものである。値は linux の標準性能分析ツールである `perf` を用い計測した。

図 7 から、`milc` では近似データの分離によるキャッシュ参照数、キャッシュミス数の変化はほぼない。これは図 4 に示したようにソースコードの変更が `malloc` を `bmalloc` 置き換えるのみであり、メモリ上のデータレイアウトが変化しないからだと考えられる。従って `milc` では近似デー

タの分離によるデータアクセスパターンの変化はなく、提案手法によって CPU キャッシュの影響を無視した近似データへのアクセス回数を正確に検出できる。一方 mcf ではキャッシュ参照数は約 1.3 倍、キャッシュミス数は約 2.3 倍に増加している。これは 図 5 に示したように構造体中の一部のメンバのみを bmalloc で確保するよう変更したためデータアクセスの局所性が変化したと考えられる。従って mcf では近似データへのアクセス回数を実際よりも大きく見積もっていると考えられ、将来的にはこれを補正する手法が必要である。

以上から提案手法の適用可能性について以下が言える。

(1) milc のようにエラー混入を許すデータ全体を malloc で確保する場合、提案機構で近似データへの総アクセス数を正確に計測できる。類似のアプリケーションには例えば大きな行列を malloc しそれに遷移行列を掛ける物理シミュレーションなどが考えられる。

(2) mcf のように構造体に近似データと厳密データが混在する場合、現状の提案機構で計測する近似データへの総アクセス数は実際より多く、補正する手法が必要である。類似のアプリケーションには例えば極度に偏りのある実世界データのグラフ構造を vertex centric モデル [7] で計算する場合などが考えられる。

## 6. 議論

### 6.1 PEBS を用いたアクセス位置の推定

本稿では、実装を単純化するため以下の二つの仮定を置いて実験を行った。

(1) 近似データへのアクセスは全ビットに均等に発生する。この仮定により近似データへの IO 量を近似データのサイズで割れば 1 ビット辺りのアクセス回数になる。

(2) 近似データのエラー発生は 1 アクセスごとに均等な確率で起こる。この仮定によりあるビットのエラー発生率は当該ビットへのアクセス回数のみから見積もれる。

一方、実際のアプリケーションでは近似データへのアクセスは一様ではないため、近似実行の影響の調査を正確に行うには各メモリセルごとのアクセス回数が必要である。また DRAM の IO 待機時間を削減することによるエラーの発生も、DRAM の内部構造のために単純にアクセス回数のみでは決定されない。例えば、同一の column に連続してアクセスが起こった場合、それらのアクセスに対する row アクチベーションは一回のみであるためエラー発生確率は 1 回のアクセスと同じである。

ハードウェアエミュレータやメモリトレースを利用せず上記（1)、(2）を考慮した近似実行のアプリケーションへの影響を知るため、今後は PEBS（Precise Event Based Sampling）[8] の利用を検討する。PEBS は Intel の CPU に搭載された性能カウンタの拡張であり、通常の性能カウンタより低オーバーヘッドである等の利点を持つ。

PEBS の特徴のうちロード命令の対象アドレスを取得する機能を用いて近似データへのアクセスを細粒度に検知する。ラストレベルキャッシュのミスを計測する性能カウンタと組み合わせて利用することで、キャッシュの影響を考慮し実際に DRAM にアクセスしたロード命令のアクセス先を取得できると考えられる。我々は PEBS のオーバーヘッドを定量評価し、1 サンプル取得毎のアプリケーション性能低下が約 250 ナノ秒と十分小さいことを示した [1]。しかし PEBS の目的はサンプリングであり、設計上全てのキャッシュミスを取得はできない。サンプリングレートが非常に高い場合にはサンプル保存中に発生した別のサンプルが取得できず取得可能なサンプル数が設定値より大きく減少するとの報告がある [11]。従って、各ビットへのアクセス数を十分に推定できかつハードウェア設計上実現可能なサンプリングレートなどの調査が必要である。

### 6.2 エラー混入によるキャッシュ無効化

提案機構ではエラー混入時に近似データへ書き込みを行うが、この時キャッシュの一貫性制御により当該データの CPU キャッシュが自動的に無効化される。これには以下の 2 点の影響がある。

(1) 提案手法ではキャッシュの効果を考慮し近似データへの IO 量を計測するが、キャッシュの無効化によって IO 量が変化してしまう。

(2) 実際の近似実行ではメモリ上で化けてしまったデータでもキャッシュ上では化けていない等が考えられ、現状の提案手法ではこの可能性を考慮できない。

これらの影響を排除するためには、近似データの一部にエラーを混入する際に以下の 2 つを行う必要がある。

(1) エラー混入によって CPU のキャッシュが無効化されたかどうかを知る

(2) 無効化されていれば、無効化されたラインを再度キャッシュに乗せる

あるデータが CPU のキャッシュに載っているかどうかを非破壊に（キャッシュの状態を変えずに）知ることは現在のハードウェアではサポートされない。アクセス履歴からキャッシュの状態を追跡する機構をソフトウェアで実装すれば可能であるが、これはハードウェアエミュレータを用いて近似実行の影響を調査する手法に近いオーバーヘッドがかかり、本研究の目指す方向ではない。考え得る展開は、キャッシュ無効化がどの程度の確率で無効化されるかを定量的に調査し、アクセス回数とその確率から計測された IO 量を補正することである。

### 6.3 異なる近似実行モデルへの対応

提案手法による近似データへのアクセス数の軽量な見積りは、本稿での近似実行モデル以外にも適用可能である。具体的には既存研究で仮定されているように DRAM のリ

フレッシュレートを下げることでリフレッシュにかかる電力を削減するモデルが考えられる。

DRAM のリフレッシュレートを下げるモデルでは、あるメモリセルに一定期間アクセスが「ない」場合にデータが一定確率で化ける（本稿ではアクセスが「ある」場合にデータが化けるとした）。このモデルでもメモリセルへのアクセスの有無を CPU キャッシュの影響を排除して軽量に見積もる必要があり、提案手法は既存のハードウェアエミュレータやメモリトレーサに対して大きく有利である。

## 7. 結論と今後の課題

本稿ではデータセンタ消費電力の増大に対応するため、近似実行をメモリに適用する際に必要となるアプリケーションへの影響調査を高速に行う手法を提案した。近似データへの IO を厳密データから分離することでメモリトレースやハードウェアエミュレーションなしに近似データへの正確なエラー混入を行う手法を示した。評価の結果、提案手法は既存研究に比較して近似実行のアプリケーションへの影響を数百倍高速に見積もれることが分かった。

本稿では全ビットへの均一なアクセスの仮定、近似データの分離によるアクセスパターンの変化に起因する IO 量の増大など改善すべき課題が残されている。また本稿で実装した `bmalloc` を用い、実際に NUMA ノード 1 に電荷チャージ待ち時間を短縮した DRAM を搭載すれば第 3 章で示した近似実行が実際に実現可能であり、改変したハードウェアや FPGA 等を用いて実際に本稿で示した近似実行モデルを実現することも将来の課題である。

**謝辞** 本成果は、国立研究開発法人新エネルギー・産業技術総合開発機構（ＮＥＤＯ）の委託業務の結果得られた。

### 参考文献

[1] Akiyama, S. and Hirofuchi, T.: Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis, *International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pp. 1–8 (2017).

[2] Carbin, M., Misailovic, S. and Rinard, M. C.: Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware, *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pp. 33–52 (2013).

[3] 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).

[4] Chen, D., Ye, C. and Ding, C.: Write Locality and Optimization for Persistent Memory, *International Symposium on Memory Systems (MEMSYS)*, pp. 77–87 (2016).

[5] ELPIDA: Low Power Function of Mobile RAM - Partial Array Self Refresh (PASR), (online), available from ⟨`https://www.micron.com/~/media/documents/products/technical-note/dram/e0597e10.pdf`⟩ (2005).

[6] Esmaeilzadeh, H., Sampson, A., Ceze, L. and Burger, D.: Architecture Support for Disciplined Approximate Programming, *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 301–312 (2012).

[7] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D. and Guestrin, C.: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 17–30 (2012).

[8] Intel Corporporation: Intel 64 and IA-32 Architectures Software Developer Manuals, (online), available from ⟨`https://software.intel.com/articles/intel-sdm`⟩ (2016).

[9] Koshiba, A., Hirofuchi, T., Akiyama, S., Takano, R. and Namiki, M.: Towards Write-back Aware Software Emulator for Non-Volatile Memory, *IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6 (2017).

[10] Kultursay, E., Kandemir, M., Sivasubramaniam, A. and Mutlu, O.: Evaluating STT-RAM as an energy-efficient main memory alternative, *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 256–267 (2013).

[11] Larysch, F.: Fine-Grained Estimation of Memory Bandwidth Utilization, Master thesis, Karlsruhe Institute of Technology (KIT), Germany (2016).

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

[13] Liu, S., Pattabiraman, K., Moscibroda, T. and Zorn, B. G.: Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning, *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 213–224 (2011).

[14] Naftaly S.: Pin - A Dynamic Binary Instrumentation Tool, (online), available from ⟨`https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`⟩ (2012).

[15] Park, H., Baek, S., Choi, J., Lee, D. and Noh, S. H.: Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-core, Multi-bank Systems, *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 181–192 (2013).

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

[17] 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, *International Symposium on Computer Architecuture (ISCA)*, pp. 349–360 (2014).

[18] Zhang, Z., Zhu, Z. and Zhang, X.: A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality, *International Symposium on Microarchitecture (Micro)*, pp. 32–41 (2000).