システムソフトウェアによるアプリケーション実行の自動効率化

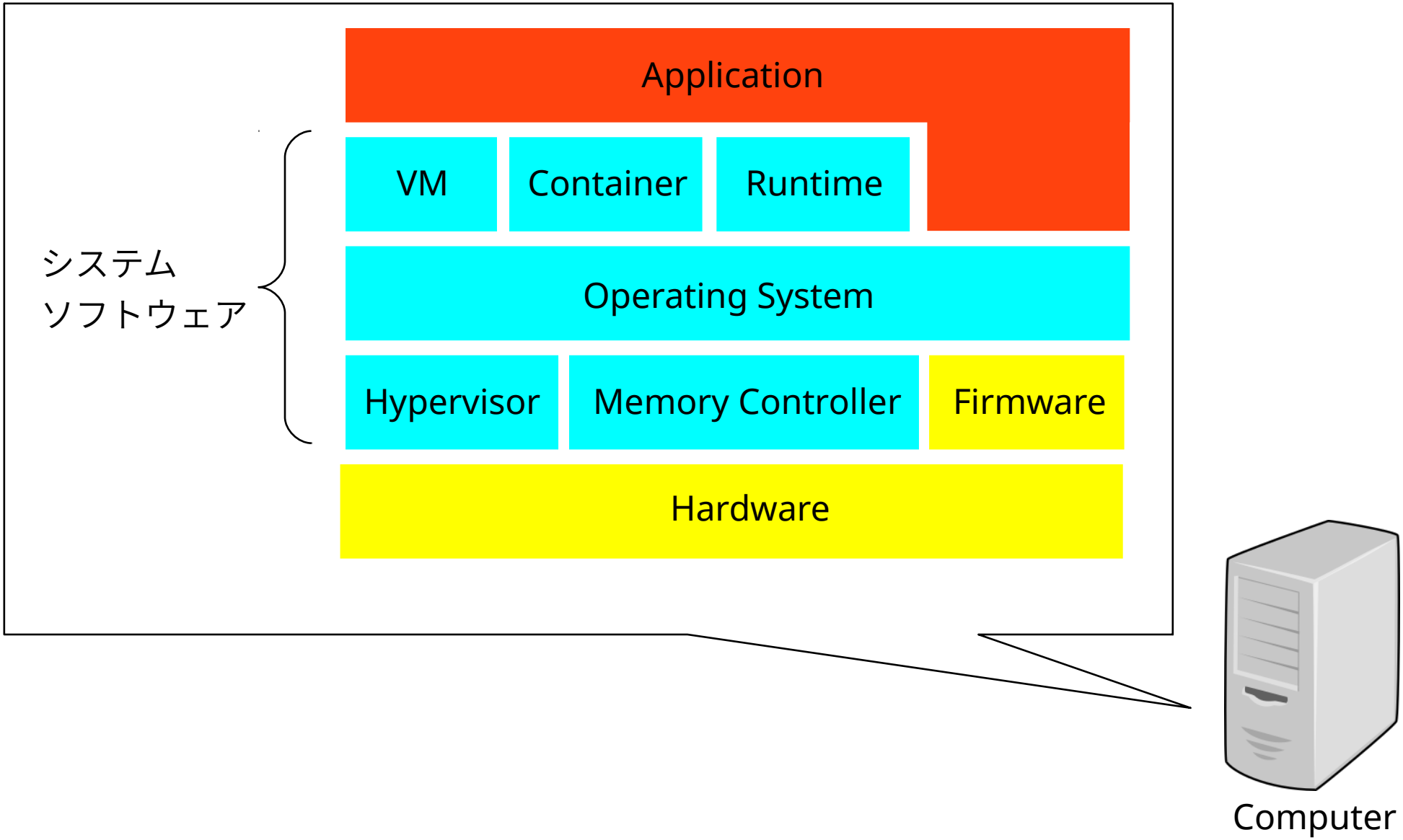
2017 年 12 月 22 日 @ 東大創造情報学専攻

産業技術総合研究所 人工知能研究センター
穐山 空道
s.akiyama@aist.go.jp

自己紹介

- 穐山 空道（あきやま そらみち）　http://www.soramichi.jp/
- **2010 / 3**　京都大学工学部情報学科　卒業
- **2015 / 3**　創造情報学専攻　博士課程修了
  - D2 のはじめに秋葉原から I-REF へ移動
- **2015 / 4 – 2015 / 10**　NTT 研究所　研究員
- **2015 / 11 – 現在**　産総研 AI センター　研究員
- 最近の興味
  - AI アプリケーションの性能予測・改善
  - メモリ周辺（DRAM, 不揮発性メモリ, 省電力・省スペース）

システムソフトウェア

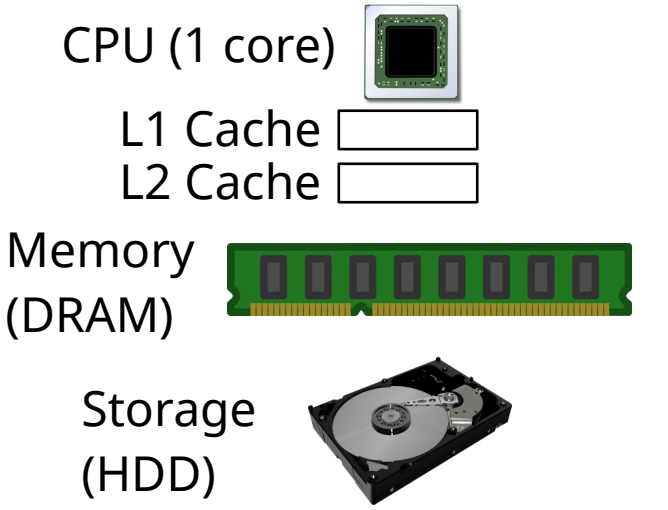

Application
VM
Container
Runtime
システム
ソフトウェア
Operating System
Hypervisor
Memory Controller
Firmware
Hardware
Computer

はじめに

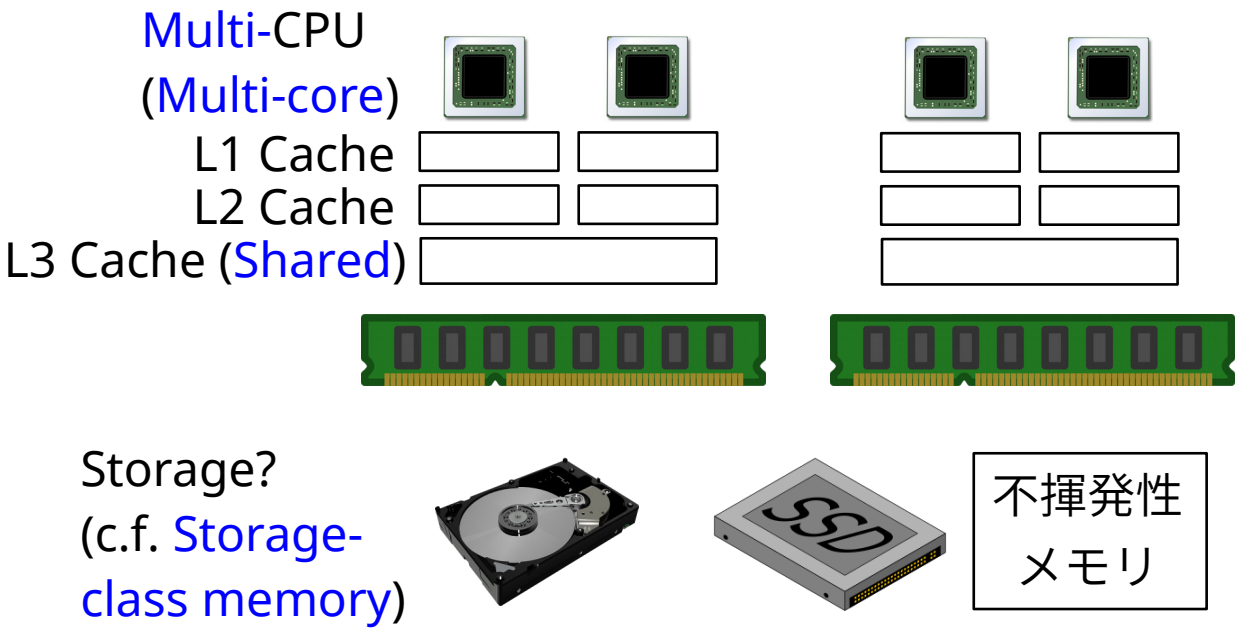
- AI センターでシステムソフトウェアを研究する意味
- アプリ側の改善だけでは十分な性能が得られない
  - 性能：速度（ faster is better ）、消費電力（ lower is better ）
- 1) システムの内部が刻々と進化
  - 内部構造の複雑化
  - 内部の性能バランスの変化
- 2) ユーザから見たシステムと実際のシステムのギャップ

システム内部の変化 (1/2)

- システム内部の構成の複雑化
  - 並列度（ CPU 数、コア数）の増加
  - メモリ帯域向上のためのキャッシュ階層の増加
  - 不揮発性メモリなど新しいデバイスの利用



10 数年前のコンピュータ

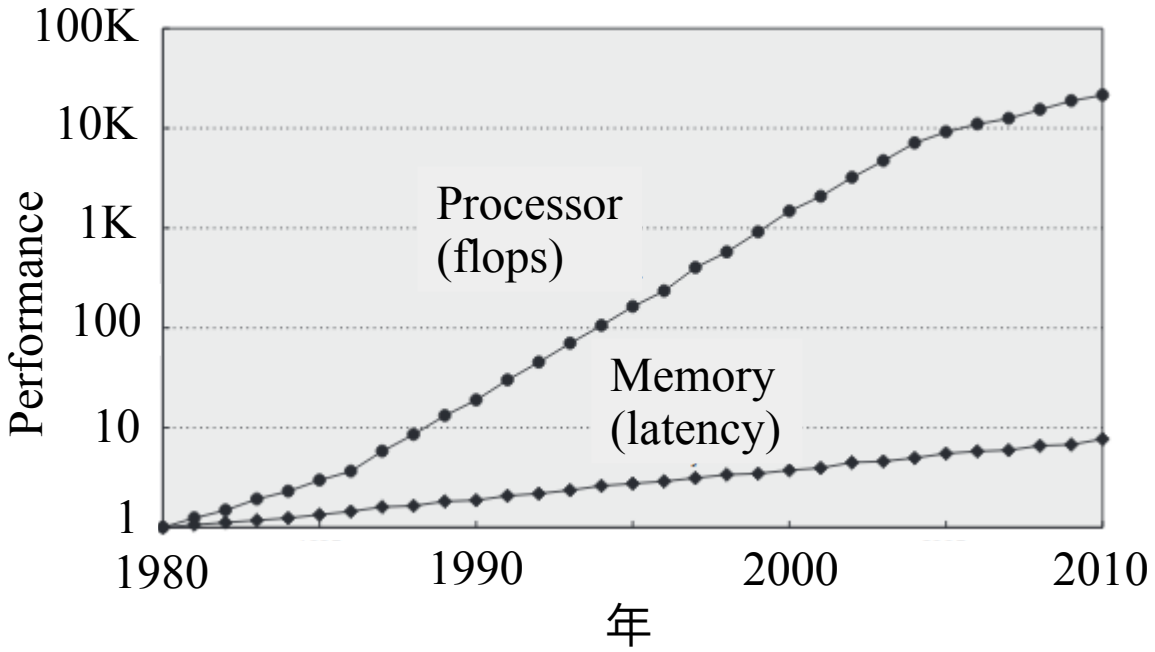


現在のコンピュータ

システム内部の変化 (2/2)

- システム内部の性能バランスの変化
  - マシン内の各コンポーネントが同一の速度で進化しない

  → 同一のソースコードのアプリでもボトルネックが変化



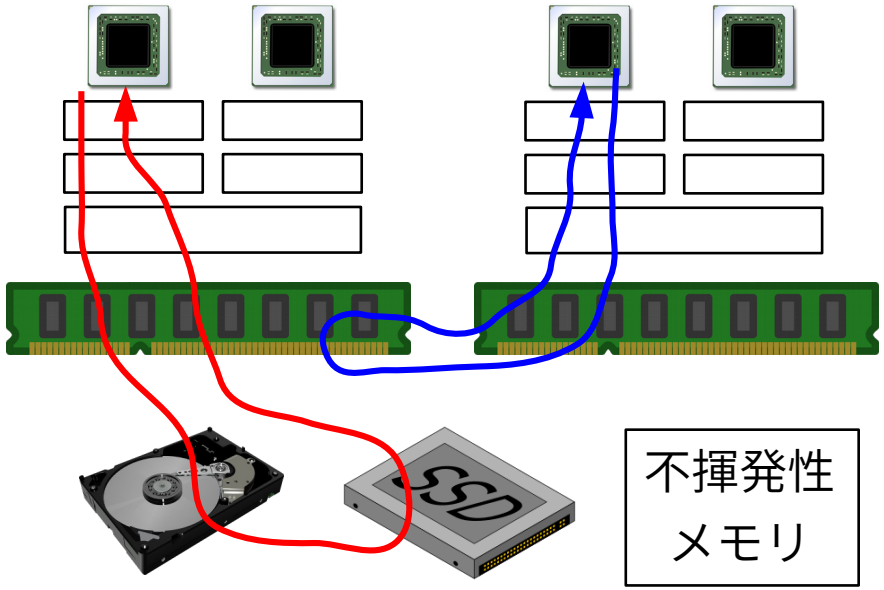
CPU(flops) とメモリ（レイテンシ）の性能向上
(*) より引用

- CPU とメモリで性能向上の速度が大きく違う（注：縦軸は対数）
- Flops は並列化で上げられるがレイテンシは並列化で改善しない
- 特にメモリレイテンシは近年ほとんど改善していない

(*) J. L. Hennessy and D. A. Patterson, “Computer Architecture, Fourth Edition: A Quantitative Approach”, Morgan Kaufmann Publishers

ユーザとシステムのギャップ

- ユーザはアルゴリズムを（超）高級言語で実装
  - 生産性のため（システムを「使いこなす」ことは本質ではない）
  - `c = aᵀb  →  c = numpy.dot(a, b)`
- 実際のシステムの動き



- ユーザからは隠蔽されているが様々なことが起こっている
- いつでも・どこでも同じことが起きるとは限らない
- ユーザが全て考慮して最適なプログラムを書くことは不可能
- HPC では C や fortran を使いアプリのライフタイムが長いのでやや可能?

システムソフトによるアプリの効率化

- システムの内部構成・性能バランスが変化
- ユーザは生産性第一（システムの使いこなしは非本質的）

システムソフトウェアがアプリケーションの性能を自動的に最大限引き出すことが重要

- そのために …
  - システムの内部構成や性能バランスの変化がアプリケーションにどう影響するかを理解する研究（今ここ）
  - 得られた知見を用いたシステムを創る研究

産総研 AI センターでの私の研究

★ 新たなデバイス（不揮発性メモリ）の上でのアプリケーションの性能予測
- A. Koshiba, T. Hirofuchi, S. Akiyama *et al.*, “Towards Write-back Aware Software Emulator for Non-Volatile Memory”, *NVMSA’17*, pp. 1 – 6

★ Approximate Computing の計算結果への影響推定 （強化し Micro に投稿予定）
- 穐山ほか, “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” , *SWoPP’17*, pp. 1 – 9
- 濱田, 穐山ほか, “Approximate Computing での NaN の除去による継続的な実行” , *ComSys’17* , poster

★ 異なる性能バランスのマシン上でのアプリケーションの性能予測
- S. Akiyama *et al.*, “Performance Prediction of Memory Access Intensive Apps with Delay Insertion: A Vision”, *CloudCom’16,* pp. 492 – 496

プロファイルをマイクロ秒単位で可能にし、各データ（クエリ、パケット等）ごとの処理時間を詳細分析
- S.Akiyama *et al.*, “Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis”, *ROSS’17 (colocated with HPDC’17),* pp. 1 – 8
- S. Akiyama *et al.*, “Per-Data-Item Profiling using a Hardware-Based Sampling Mechanism”, *under review for CCGrid ‘18*, 10 pages

アクター型分散システムの性能最適化
- A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama *et al.*, “Optimizing Distributed Actor Systems for Dynamic Interactive Services”, *EuroSys’16*, pp. 1 – 15

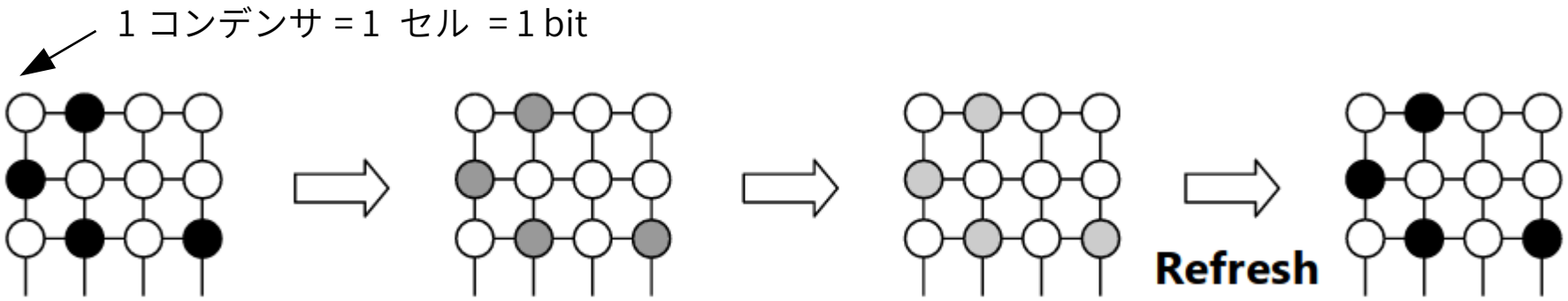
★ 本日の内容（その他は少し系統が違うので本日は割愛）

新たなデバイス（不揮発性メモリ）上でのアプリケーションの性能予測

A. Koshiba, T. Hirofuchi, S. Akiyama *et al.*, “Towards Write-back Aware Software Emulator for Non-Volatile Memory”, *NVMSA’17*, pp. 1 – 6

背景 (1/2)

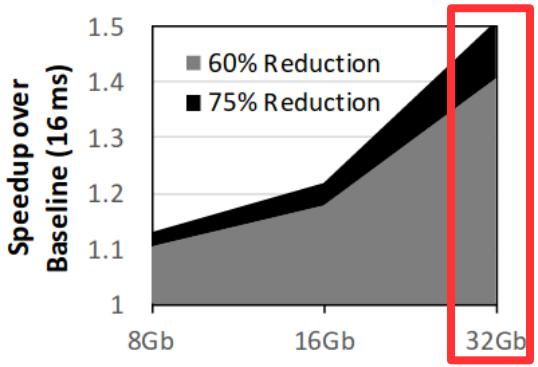
- 現在実運用されているマシンのメモリは DRAM
  - 各セルはコンデンサ、徐々に電荷が抜ける → リフレッシュ必須



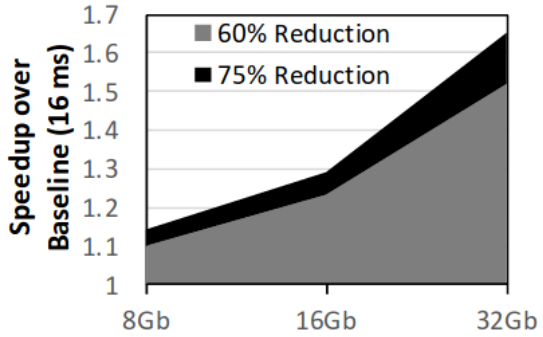
- AI 、ビッグデータ処理の隆盛でメモリ容量増加
  - 産総研 ABCI では 1 ノードあたり 256GB の予定
  - DRAM 高密度化 → セルの小型化 → リフレッシュ頻度の増加

背景 (2/2)

- リフレッシュはオーバーヘッドが大きい
  - リフレッシュ中の row （一行に並んだセル）にはアクセス不能
  - リフレッシュするのに電力がかかる
- リフレッシュ頻度を 1/4 に → アプリ性能が数 10% 向上
  - 逆に言うと：リフレッシュが頻繁すぎアプリに悪影響



(a) Single-core



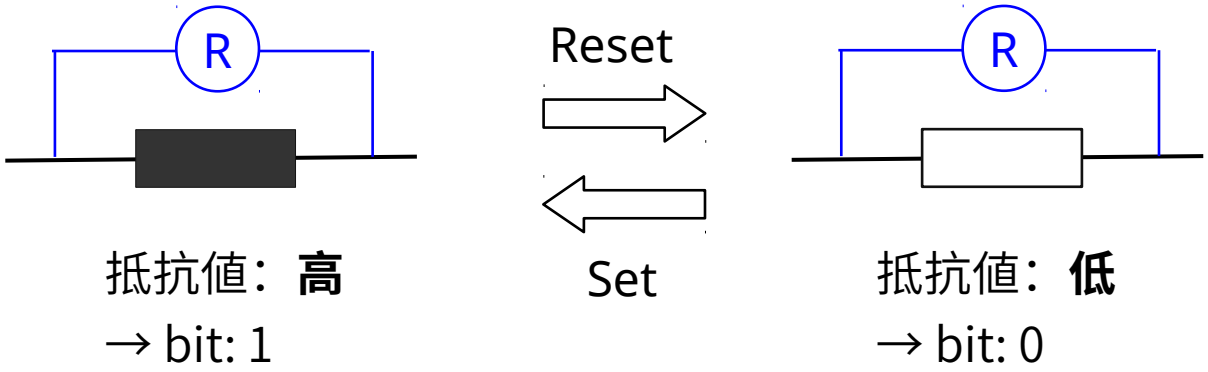
(b) Four-core

- [S. Khan *et al.*, Micro'17] から引用
- リフレッシュ回数が 75% 減ると性能が最大 5 割向上

DRAM に変わる新たなデバイスへの移行が近いと言われている

不揮発性メモリ

- 電荷以外の不揮発な物理的特性によってデータを表現
  - 結晶構造など
  - リフレッシュ電力が不要で省電力・大容量化に期待



- 例
  - PCM (Phase Change Memory)： 結晶構造の変化で抵抗値が変化
  - ReRAM (Resistive RAM)： 金属酸化物の状態変化で抵抗値が変化

不揮発性メモリと DRAM の違い

- 「機能」は同じ
  - データを高速に保持する
  - システムソフトウェアで隠蔽すれば同じプログラムが動く
- 「性能」の特性は大きく異なる
  - 消費電力：不揮発性メモリの方が平均では少ない
  - 速度：特に PCM では読み書きのレイテンシが不均一
    - DRAM ではほぼ読み書きのレイテンシが同じ

	2013	2026
Write / Read レイテンシ比	Write が 5 倍遅い	Write が 3 倍遅い

* ITRS のレポートより

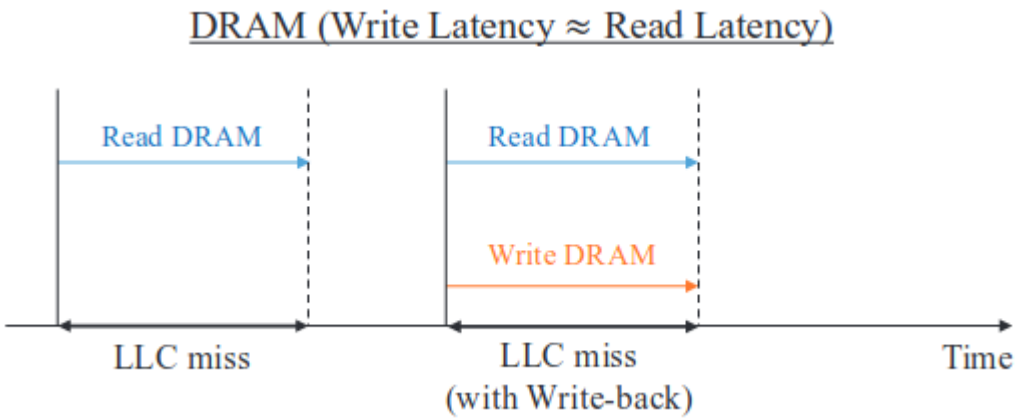
アプリケーション性能への影響

- アプリの性能が DRAM と不揮発性メモリで変化する
  - 読込が多： DRAM と不揮発性メモリで近い性能
  - 書込が多：不揮発性メモリ上では性能劣化が予想される
- 性能の変化を実デバイスなしで予測することが重要
  - 不揮発性メモリが実用化された際に、どこがシステムのボトルネックとなるか?
  - 不揮発性メモリが実用化された際に、どんなシステムソフトウェアが必用か?

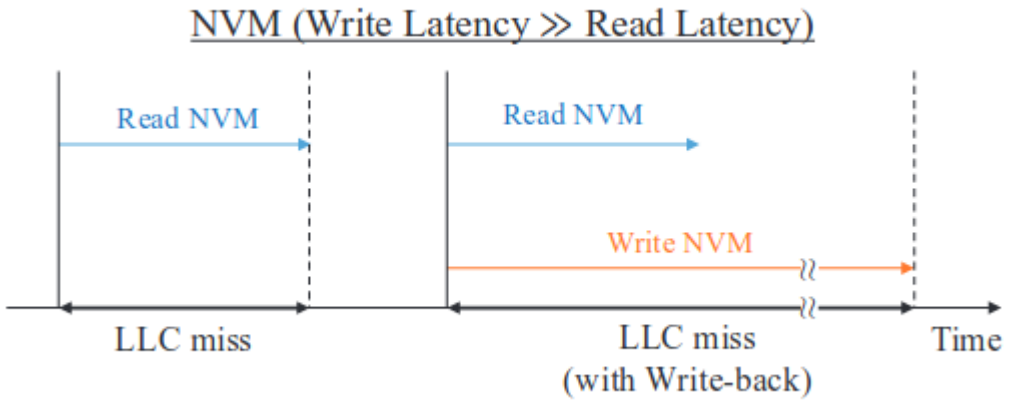
不揮発性メモリ上でのアプリの性能を予測する手法が必用

性能への影響の予測

- 実デバイスなしでの予測は難しい
  - CPU にキャッシュがあるのでコードの静的解析では予測不能
  - 実際に実行してみることが必用



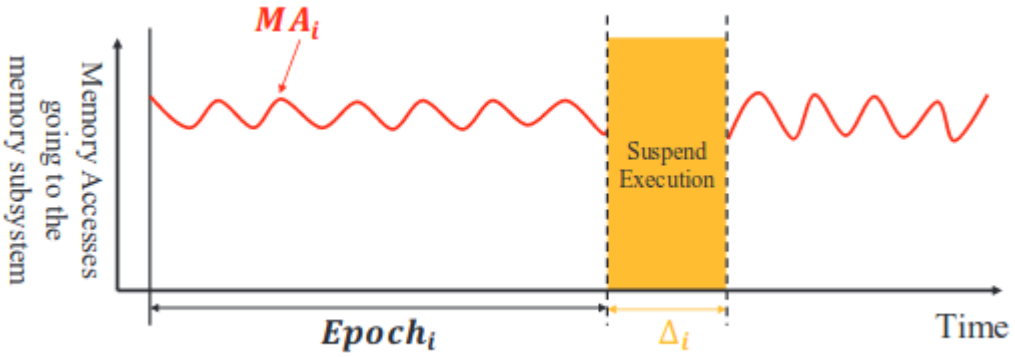
LLC (Last Level Cache 、つまり L3) ミスの DRAM 上でのレイテンシ。読込と書込が同速度なので書込があってもなくても同じ。



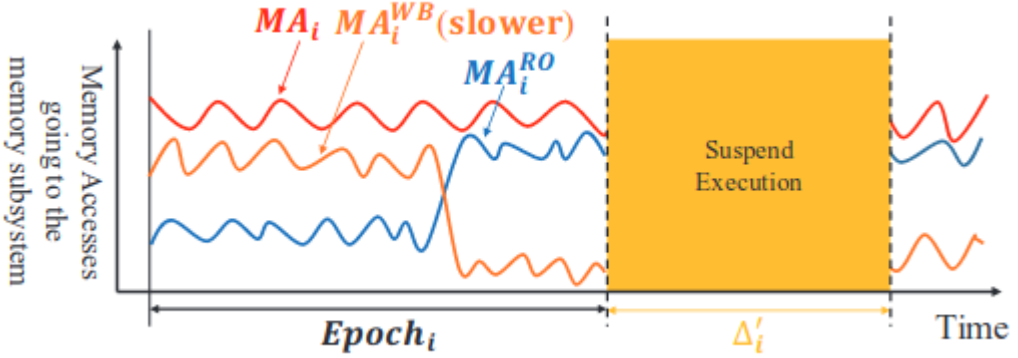
LLC ミスの不揮発性メモリ上でのレイテンシ。書込が遅いので書込の有無で速度が変わる。

不揮発性メモリ利用時の性能予測

- 基本アイディア
  - メモリレイテンシの影響はキャッシュミスの回数に比例（既知）
  - 以下を区別して数える（ new ）
    - 読み込みのみのキャッシュミス（ MA^{RO} ← RO: Read Only ）
    - 書き込みを伴うキャッシュミス（ MA^{WB} ← WB: Write Back ）
  - それぞれに比例してアプリをしばらく止める



既存手法：キャッシュミス回数に比例して sleep させる



提案手法： MA^{RO} と MA^{WB} で異なる係数を用いる

不揮発性メモリ利用時の性能予測

- 読み込み / 書き込みのキャッシュミスの区別
  - マシンをクロックごとにシミュレーション：遅い（１秒のシミュレーションに数時間）→ 実用的なアプリに適用不能
  - CPU の性能カウンタ：「全てのキャッシュミス」しか計測できず
    → Read と Write を区別できない
- メモリコントローラから取れる情報を利用
  - あるプロセス i が起こす MA^{WB} の数

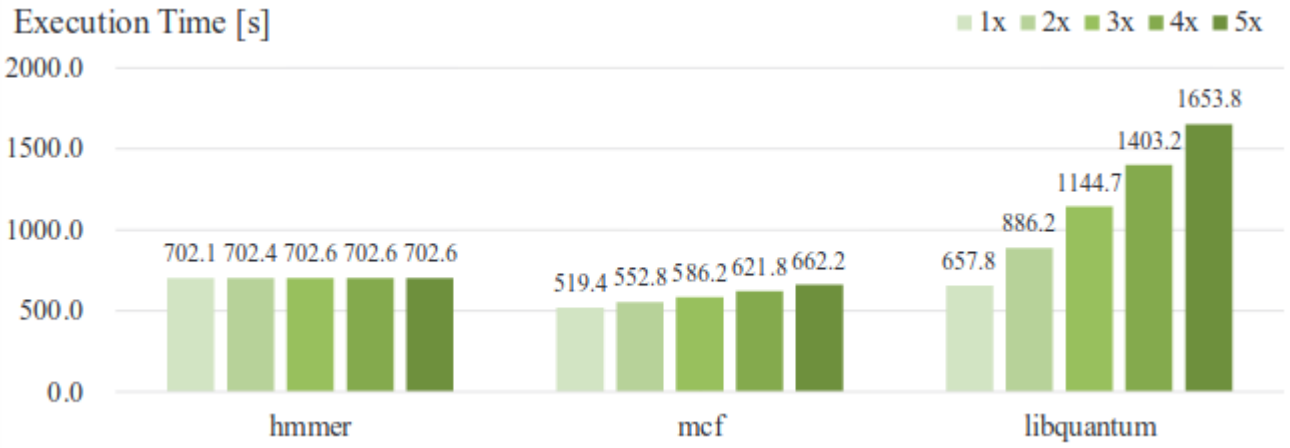
    == 全メモリ書き込み回数

    × プロセス i のキャッシュミス数 ÷ 全キャッシュミス数

メモリコントローラから取得　各コアから取得

評価実験（ 1/2 ）

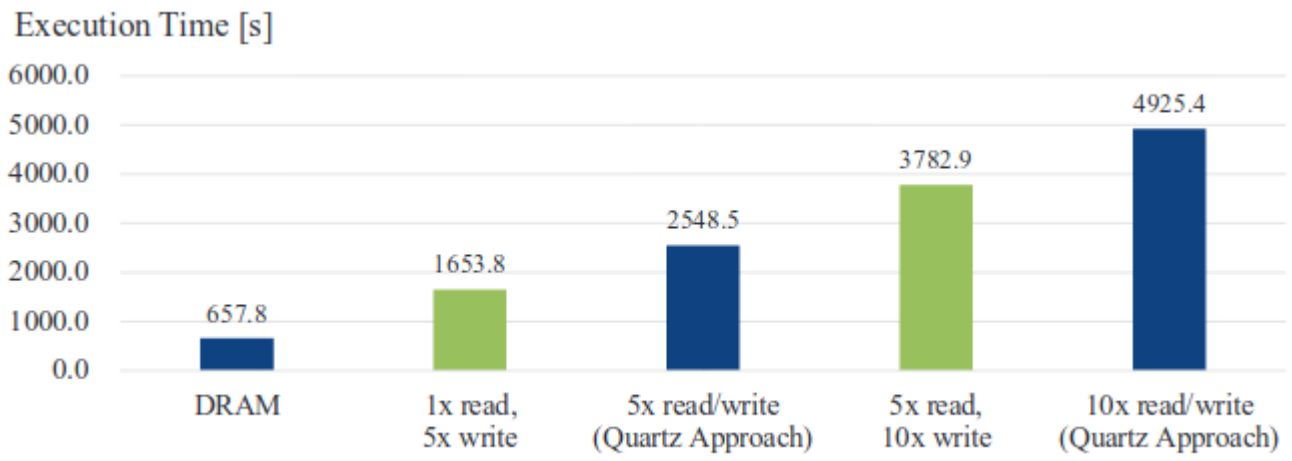
- 実験 1
  - SPEC CPU ベンチマークの性能変化を推定
    - hmmer: CPU intensive
    - libquantum: Memory intensive
    - mcf: in the middle
  - Write が Read より N 倍遅いと仮定（ N= 1, 2, 3, 4, 5 ）



- アプリケーションごとに異なる性能変化が推定可能
→ 不揮発性メモリによる影響を事前に推定・対処可能
- アプリの特性から考えて妥当な結果（注： ground truth は存在しない）

評価実験（ 2/2 ）

- 実験 2
  - 既存研究（ Read と Write と区別しない）との比較
    - 実験 1 で性能変化が大きかった libquantum を私用
  - 既存研究では Read と Write が両方 N 倍遅い、は可能



- 既存手法では不揮発性メモリ上で実行した場合の性能変化を正しく推定できない

既存： Read と Write が同じだけ遅い　提案： Write だけが遅い

関連研究

- 不揮発性メモリの Cycle accurate simulator [M. Poremba et al., VLSI'12][X. Dong *et al.*, IEEE TCAD]
  - クロックごとに何が起こるかをシミュレーション

    → 正確なシミュレーション可能（という主張）
  - 非常に遅い　実機の数百倍〜千倍
  - 実は CPU のモデルが単純なのでそう正確ではない?
- Read/Write を区別しない類似手法 [H. Volos *et al.*, Middleware'15]
  - 性能の見積もり誤差が大きい（実験 2 ）

Key Takeaways

- 広く使われる DRAM はリフレッシュのオーバーヘッドが大きい → 不揮発性メモリへの移行が近いと言われている
- 不揮発性メモリは性能特性が DRAM と異なる → アプリにどう影響するか事前に知ることが大切
- 提案手法では読み書きのレイテンシの違いを考慮することで正確・高速な性能予測を可能にした

Approximate Computing の計算結果への影響推定

穐山ほか , “ 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” , *SWoPP’17*, pp. 1 – 9

濱田 , 穐山ほか , “Approximate Computing での NaN の除去による継続的な実行” , *ComSys’17* , poster

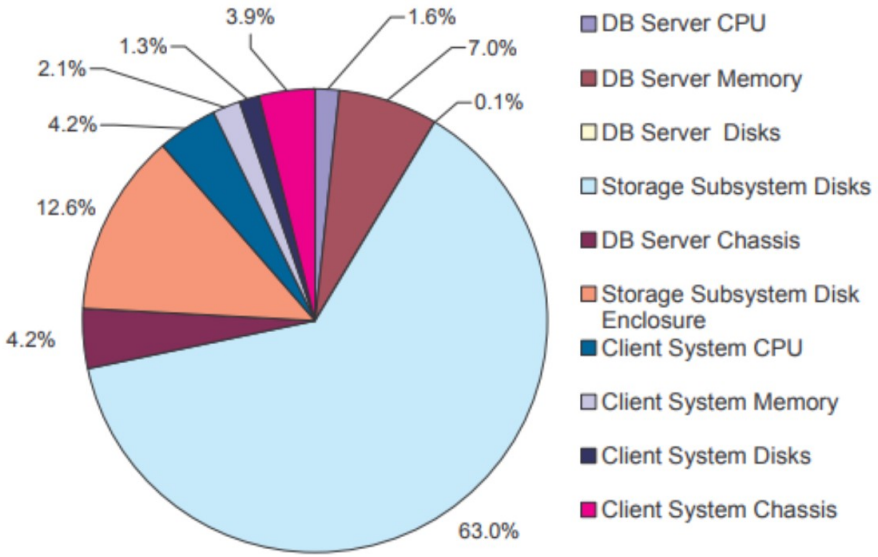
背景

- AI 、ビッグデータ処理 、大規模数値計算、…

  → 大量のメモリを利用・メモリの消費電力割合が増大

- メモリの消費電力がマシン全体の数割（と言われている）

右図： 2007 年時点でエンタープライズ DB サーバの消費電力の 7% がメモリ（現在ではさらに増加? ）



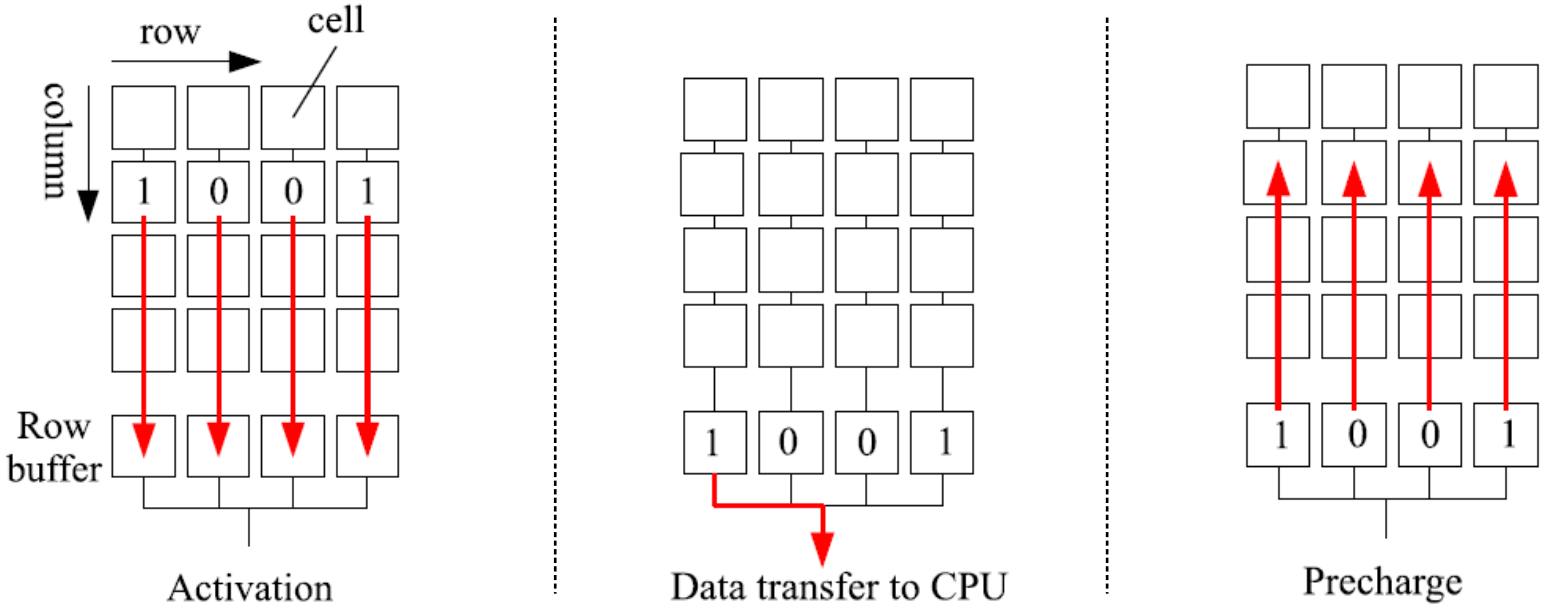
- メモリの消費電力を削減する取組
  - 不揮発性メモリの利用（既に説明）
  - 既存の DRAM を省電力で使用する

DRAM の電力削減

- リフレッシュの回数を削減（既存研究多数）
- 電荷のチャージを少しさぼる ← こちらに着目
  - [K. Chang *et al.*, SIGMETRIC'16] が初めて?
- メリット：既存の DRAM をほぼそのまま利用可
- デメリット：データにエラー（ bit flip ）が混入
  - Retention Failure: リフレッシュ間隔が長すぎて電荷が抜ける
  - Row Hammer: リフレッシュ間隔が長いとノイズ（隣の線に電流 → つられて電荷が抜ける）に弱くなる [Y. Kim *et al.*, ICSA'14]

アクチベーションとプリチャージ

- メモリ（ DRAM ） から CPU へのデータ転送時に発生

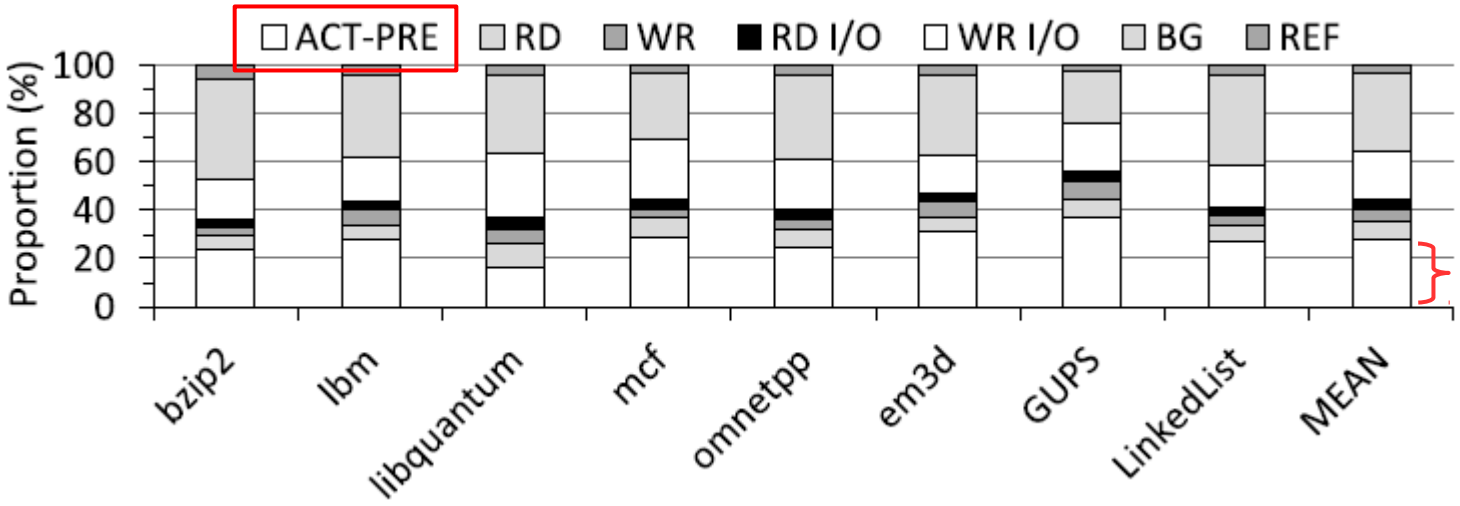


  - セル（コンデンサ）のチャージが必要 → 高消費電力

- メモリへのアクセスごとに発生
  - 本当は row buffet hit とかもあるが・・・

DRAM の消費電力

- 実際の消費電力内訳
  - アクチベーション、プリチャージが消費電力の 1 位 or 2 位



[Y. Lee *et al.*, HPCA'17] の Figure 2 より。横軸はワークロード

- 既存研究：セルのチャージ時間削減（高速化）↔ エラー率増加 [K. Chang *et al.*, SIGMETRIC'16]

本技術を消費電力削減に利用（チャージをさぼる → 電力減）

電力削減の積極性

- **Conservative な手法**：エラーが見えない程度に工夫
  - 製造のブレでエラー耐性に高低 → エラー耐性の高いのセルだけリフレッシュを削減 [S. Khan *et al.*, Micro'17]
  - アクセスされると自動的にリフレッシュ → しばらくリフレッシュしなくていい [H. Hassan *et al.*, HPCA'17]

- **Aggressive な手法**：エラーが多少起きても結果が大きく壊れなければ OK （Approximate Computing ）
  - AI：元からデータにノイズが入っている
  - 画像処理：特に結果を人が見る場合
  - 数値計算：繰り返し計算によって収束する

Approximate Computing

- 計算結果に多少の誤差を許す
  - 大幅な高速化・大容量化や普通では不可能な電力削減
  - DRAM を aggressive に使うのはその一例
- Approx Computing の例（ DRAM 以外）
  - 重い関数を NN で置き換える [H. Esmaeilzadeh *et al.*, Micro'12]
  - SSD の 1 セルに複数ビット格納（ MLC ）→ JPEG のエンコードを工夫し bit flip しても余り壊れない [Q. Guo *et al.*, ASPLOS'16]

上記から引用。普通の JPEG は圧縮されているので数ビット化けただけで画像全体が壊れる。

Approximate Computing の影響

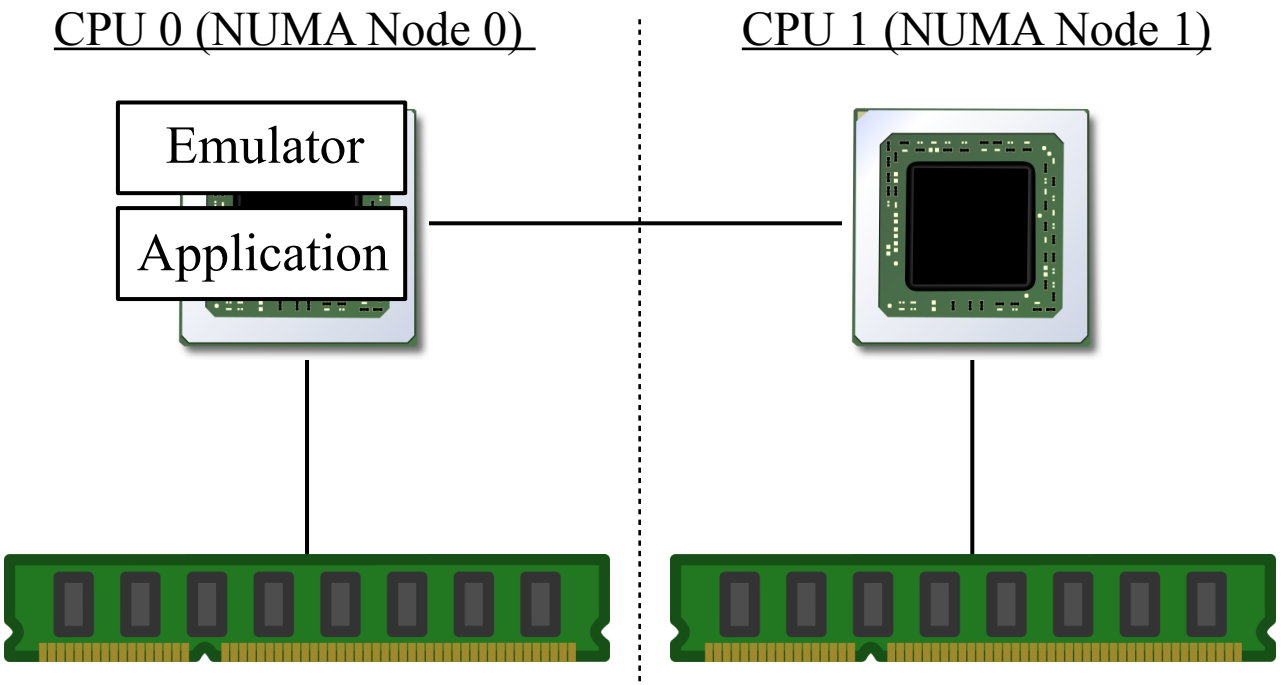
- アプリの結果への影響と電力削減率の関係が未知
  - 既知　Input: 電力削減率 → Output: ビットエラー率
  - 未知　Input: 電力削減率 → Output: アプリへの影響
  - アプリケーションによって結果への影響範囲・大きさは異なる
  - 利用シーンによって許容誤差も異なる
- 既存手法
  - Cycle accurate simulator を使用 → 非常に低速
  - { エラー率、化けてよいデータの選択 } を変えて何度も試せない

与えられたパラメータにおいて、データ化けにより
アプリの結果に発生する誤差の高速かつ定量的な推定が必用

基本アイディア

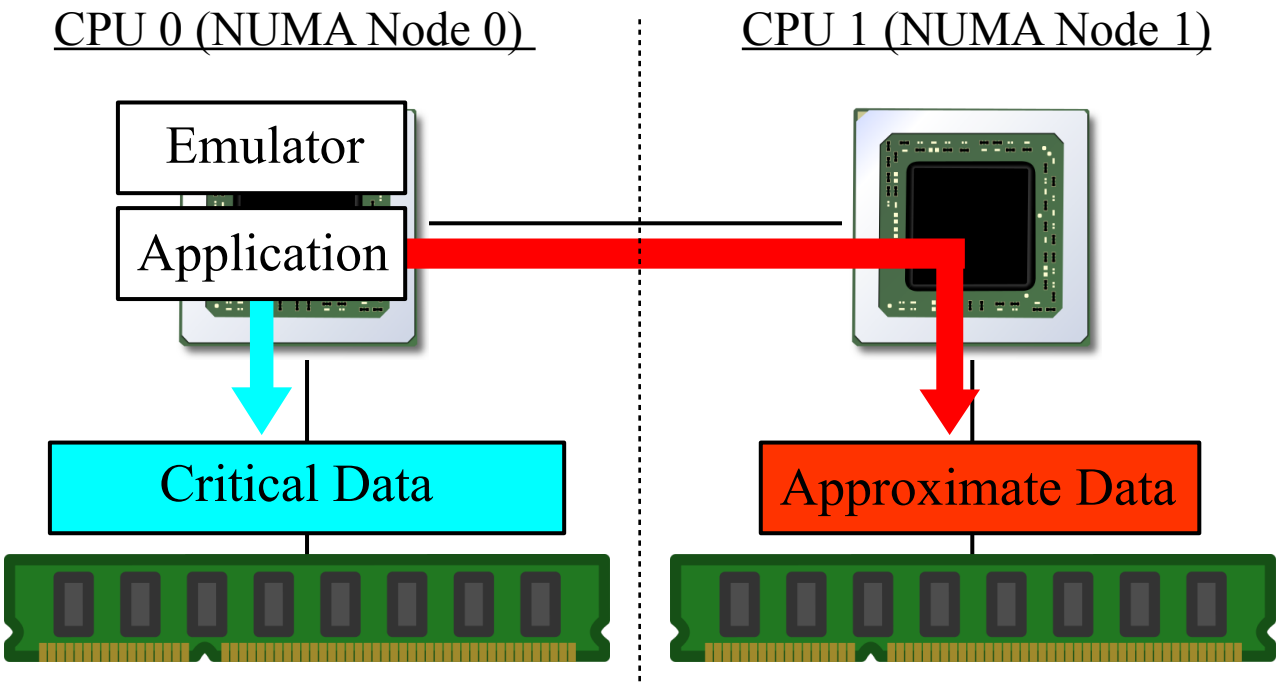
- 「発生したメモリアクセスの回数」を数えればよい
  - チャージをサボる：アクセス 1 回ごとに一定確率で bit flip
- 技術的課題
  - 「化けてもいいデータ」へのアクセスだけ数えたい
  - キャッシュの影響を排除したい
  - （注：正確なシミュレーションは遅すぎる）
- 提案手法の概略
  - 「化けてもいいデータ」と「化けてはいけないデータ」を物理的に分離
  - それぞれのメモリアクセス量を計測（キャッシュを出た後の量が測れる）

提案システム（ 1/3 ）



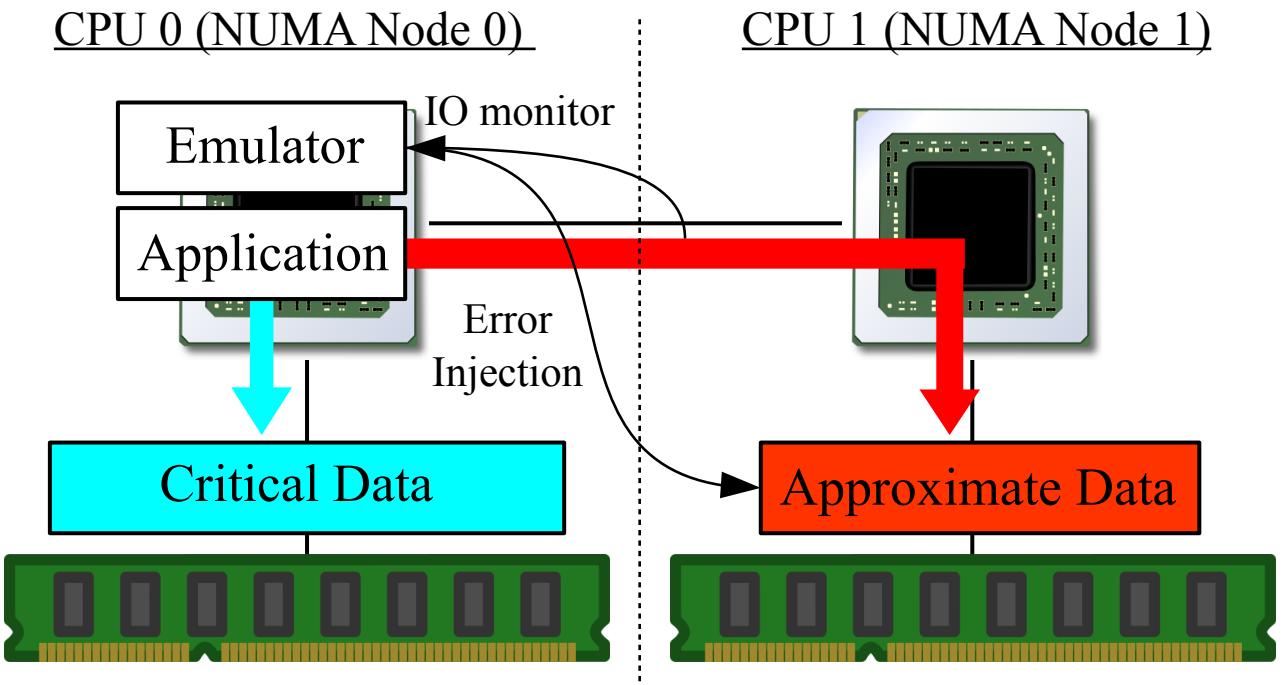
1. 対象アプリとエミュレータを CPU 0 で起動

提案システム（ 2/3 ）



1. 対象アプリとエミュレータを CPU 0 で起動
2. 厳密データを CPU 0 に、近似データを CPU 1 に配置

提案システム（ 3/3 ）



1. 対象アプリとエミュレータを CPU 0 で起動
2. 厳密データを CPU 0 に、近似データを CPU 1 に配置
3. 近似データへの IO 量を監視し、閾値に達したらエラーを混入
   - メモリコントローラのハードウェアを利用
   - キャッシュへの IO を含まず DRAM への IO のみを計測可能

評価実験

- [Q1] 近似実行を高速にエミュレーションできるか?
- [Q2] 実際のアプリケーションへの影響?
- [Q3] データ分離による誤差（ IO 量の overcount ）?
- 評価環境

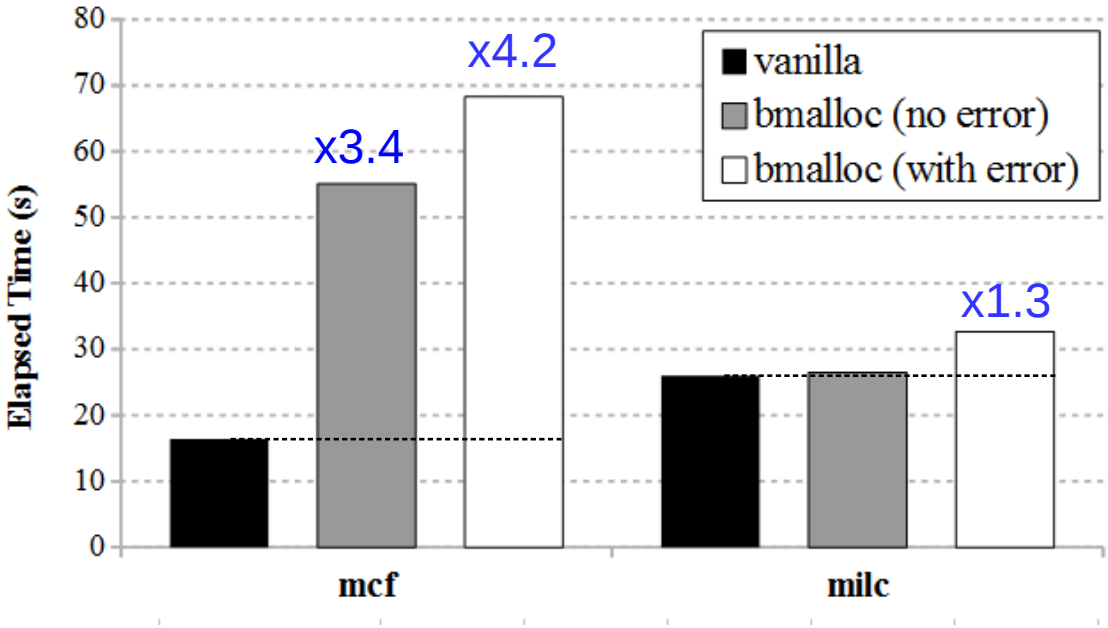
OS	Debian GNU/Linux 8 (Linux kernel 4.9)
CPU	Intel Xeon E5-2630 v4 (Broadwell) ×2
LLC size	25 MB (per NUMA node)
Memory	64 GB (per NUMA node)
Workload	milc, mcf (SPEC CPU 2006 Integer)

エミュレーションの速度

- [Q1] 近似実行を高速にエミュレーションできるか?
  - { ネイティブ実行、データ分離のみ（エラー混入なし）、データ分離かつエラー混入 } で実行時間を比較

mcf: データ分離により 3.4 倍、エラー混入により 4.2 倍に
milc: ほぼ実時間

* milc と mcf で差が出る理由は後に分析



既存手法（１回あたり４日～６日）に比べ約 480 倍の高速化

ケーススタディ

- [Q2] 実際のアプリケーションへの影響
  - mcf （探索問題）：通常実行と同一の解を出力 → 成功　異なる解を出力 → 失敗　探索が収束しない → 実行不能
  - milc （量子力学シミュレーション）：出力の通常実行との誤差

mcf

	R = 1e-11	R = 5e-11	R = 1e-10
成功率	70%	40%	10%
発生したエラー数	6 bits	21 bits	46 bits
実行可能率	100%	90%	80%

milc

	R = 1e-11	R = 1e-10	R = 1e-9
誤差	4.5e-8	8.1e-7	4.2e-6
発生したエラー数	20 bits	199 bits	2031 bits
実行可能率	100%	100%	100%

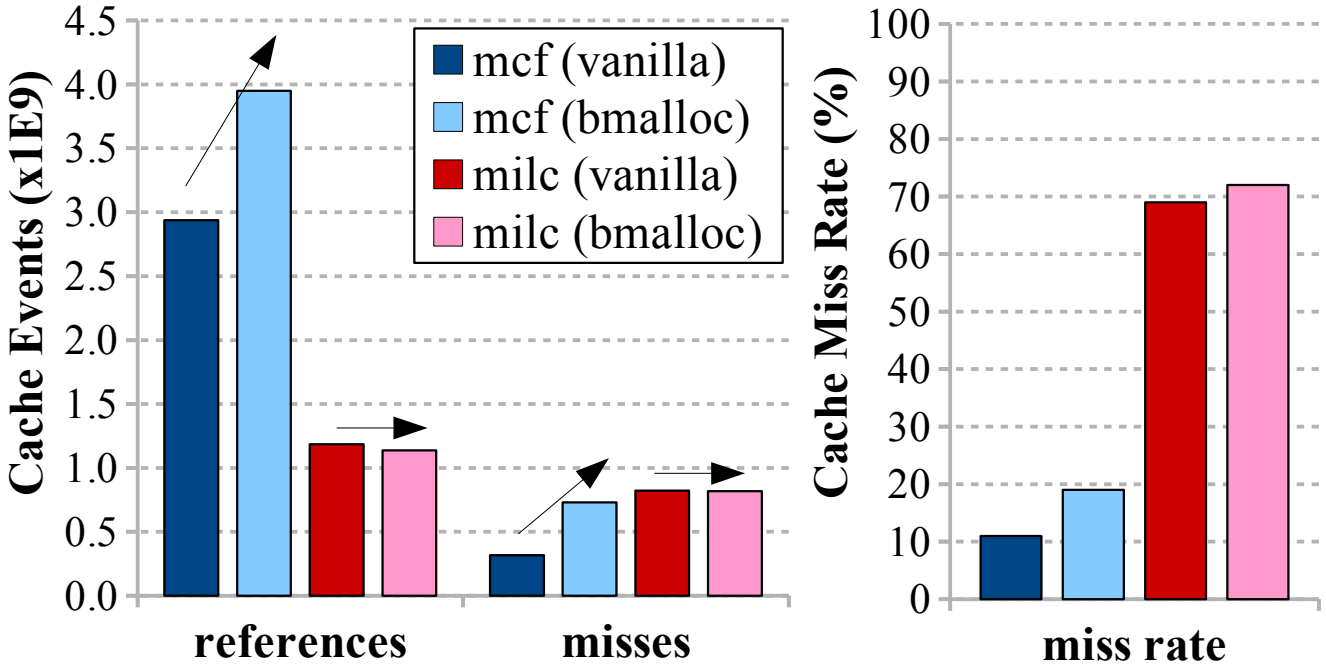
アプリへの影響はエラー率に敏感 → 高速なエミュレーションが重要

妥当性の議論（ 1/2 ）

- [Q3] データ分離による誤差（ IO 量の overcount ）？
  - 近似データの分離 → メモリ上のデータ配置が変化
  - DRAM への IO 量が増減し正確に計測できない可能性

mcf: キャッシュ参照、キャッシュミスともに増加
milc: ほぼ変化なし

→ mcf では DRAM への IO 量を多く計測している



妥当性の議論（ 2/2 ）

- milc と mcf で異なるデータ分離パターン
  - milc: 大きなメモリ領域をそのまま移動

    → データレイアウトが変化しない
  - mcf: 構造体の一部メンバのみ分離

    → データレイアウトが大きく変化
- 提案システムの他アプリへの適用可能性
  - 物理シミュレーション：大きなデータ（ e.g. 行列）を移動 → 提案システムをそのまま適用可能
  - 実世界グラフ処理（ e.g. SNS 分析）： skew を効率的に扱うためにデータ構造が複雑 → 提案手法では IO 量を overcount する

関連研究

- メモリトレース、ハードウェアシミュレータを使用
  - 広く行われる一般的な方法 (*1, *2, *3)
  - 提案する Approximate Computing モデルをシミュレータで実現
  - 実機の数百～千倍低速

- 最終結果へのエラーを静的解析で確率的に推定 (*4)
  - エラー率 R の命令が実行された回数（ N ）を推定
  - 上界しか見積もれず（最悪ケース：ループ回数不定 → N = ∞ ）

(*1) S. Liu et al., “Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning”, *ASPLOS’11*
(*2) A. Sapmson et al., “EnerJ: Approximate Data Types for Safe and General Low-power Computation”, *PLDI’11*
(*3) H. Esmaeilzadeh et al., “Architecture Support for Disciplined Approximate Programming”, *ASPLOS’12*
(*4) M. Carbin et al., “Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware”, *OOPSLA’13*

Key Takeaways

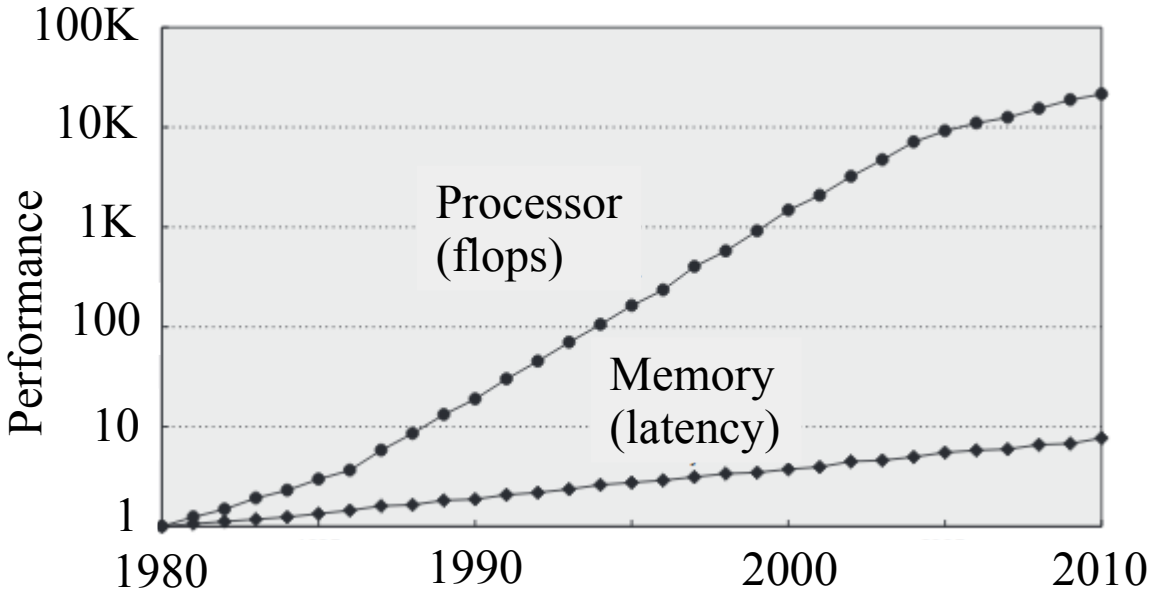
- メモリの大容量化への要求 → メモリの消費電力が増大
- 既にある DRAM をアグレッシブに利用して消費電力を減らす Approximate Computing （↔ 不揮発性メモリ）
- アプリの計算結果に生じる誤差の定量評価が重要
- 提案手法では実機をうまく使うことで高速に評価可能 → 複数のパラメータを変えて何度も試せる

異なる性能バランスのマシン上での性能を予測する

S. Akiyama *et al.*, “Performance Prediction of Memory Access Intensive Apps with Delay Insertion: A Vision”, *CloudCom’16*, pp. 492 – 496

背景

- マシン内の性能バランスが変化



- Flops（１秒あたりの浮動小数点演算数）がメモリレイテンシの何倍もの速さで増加

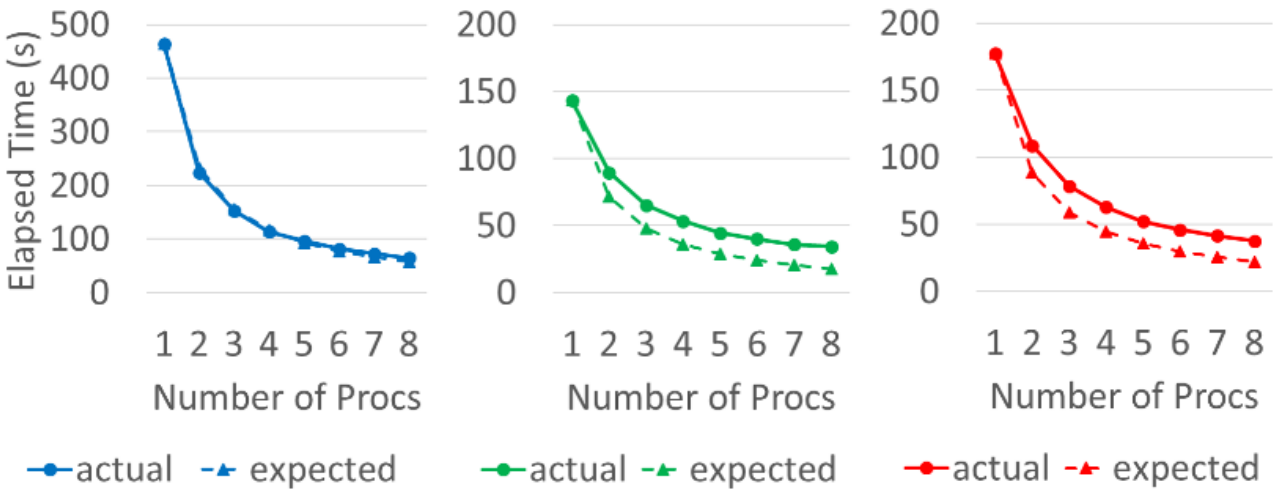
→ 新しいマシンではメモリアクセスがボトルネックになりやすい

バランスの違いによる性能変化

- Python, ALS アルゴリズムによる Matrix Factorization
  - 間隔が非一様なメモリアクセスが多発

	Machine 1	Machine 2	Machine 3
CPU	Xeon E5-2603 **v なし**	Xeon E5-2699 **v3**	Xeon E7-8863 **v3**
Mem	DDR3 (1600MHz)	DDR4 (2133 MHz)	DDR4 (2133 MHz)

⇦ 旧　　新 ⇨



Machine 1　Machine 2　Machine 3

actual: 実測値
expected: linear scale

古いマシンでは linear scale するが、新しいマシンではしていない!

なぜ困るか?

- 大規模マシン用の開発（ AI 、ビッグデータ）
  - 手元のマシンで開発・デバッグ → 大規模共有マシンで実行
  - 例：産総研 ABCI ではコンテナでパッケージしたアプリを直接実行可能になる予定（↓産総研・佐藤さんの GTC での発表）
  - https://www.slideshare.net/htsst/gtc-japan-2017-84434985
- 手元のマシンと大規模マシンでは性能バランスが違う
  - どちらかが少し古い・新しい
  - サーバ用 CPU は周波数が低い → メモリレイテンシが大きい

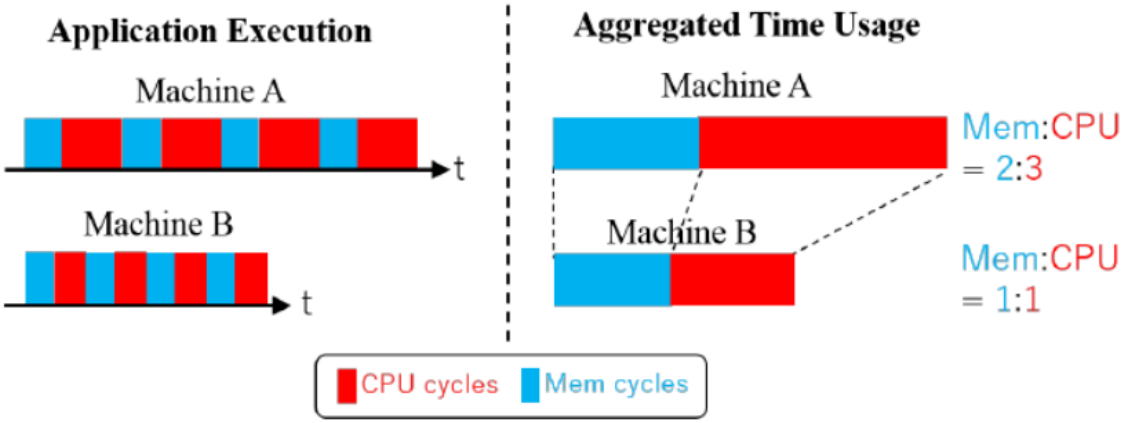
手元で高効率な実装が大規模マシンでも高効率とは限らない

メモリイテンシブアプリの性能予測

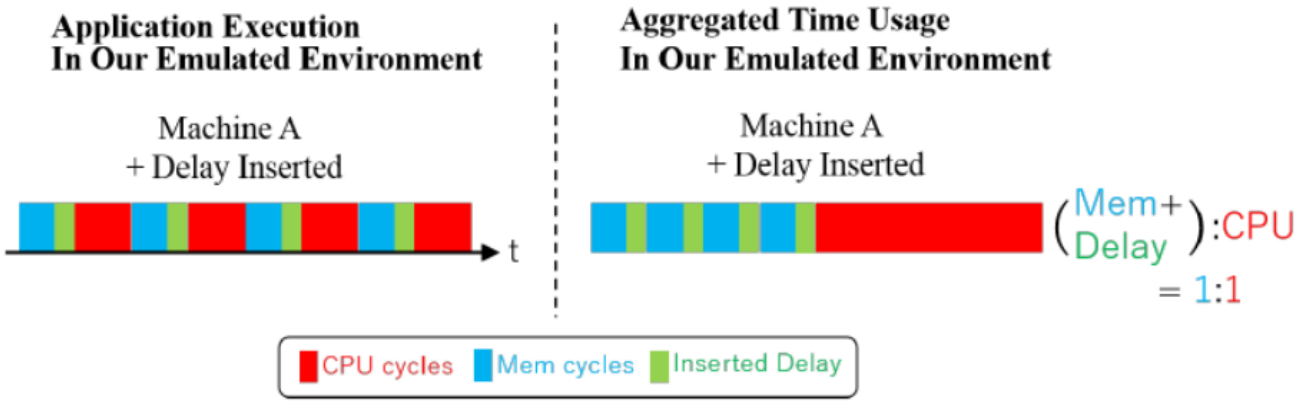
- 達成したいことは以下
- メモリイテンシブアプリのパフォーマンスが性能バランスの異なるマシンでどう変化するかを知りたい
- 手元のマシンだけ使って知りたい
  - 本番用大規模サーバ上では try and error はできない
- 各関数ごとに知りたい
  - ↔ プログラム全体でだいたいわかればよい

基本アイディア

- メモリ IO 命令を少しだけ遅延させる
  - 命令ごとに遅らせる → 関数単位で忠実な推定が可能



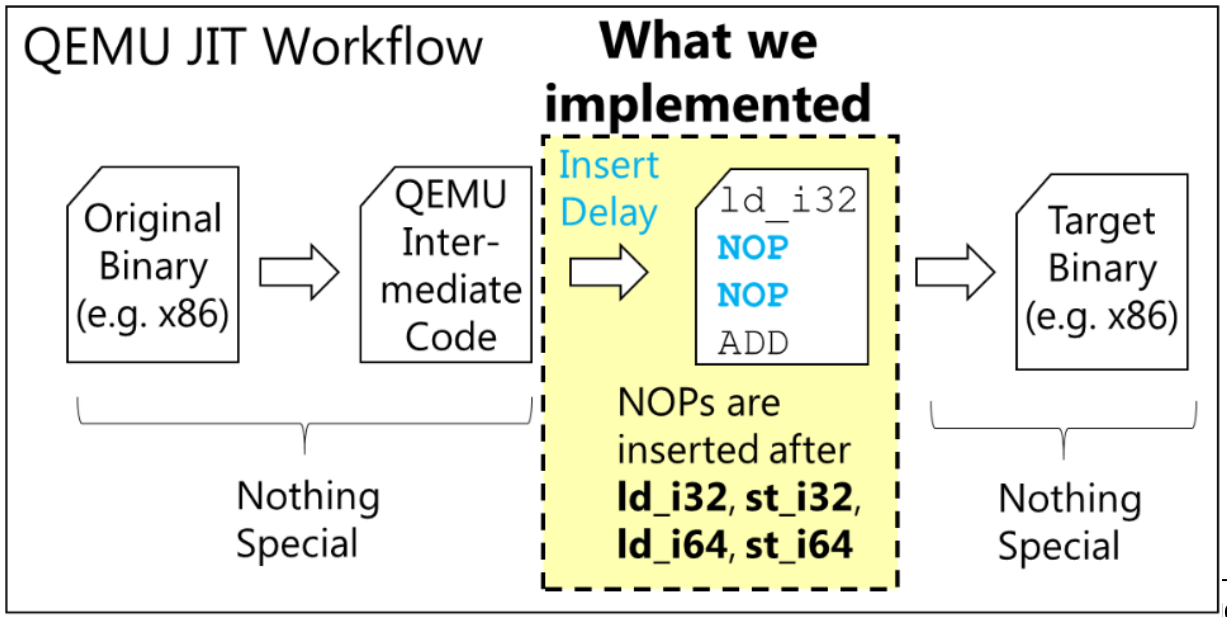
- マシン A ではメモリアクセスに全体の 2/5 の時間を使う
- マシン B ではメモリアクセスに全体の 1/2 の時間を使う（バランスが違う！）



- マシン A ではメモリアクセスを少し遅らせれば、全体の 2.5/5 == 1/2 にできる → マシン B での性能を再現可能!

提案システム

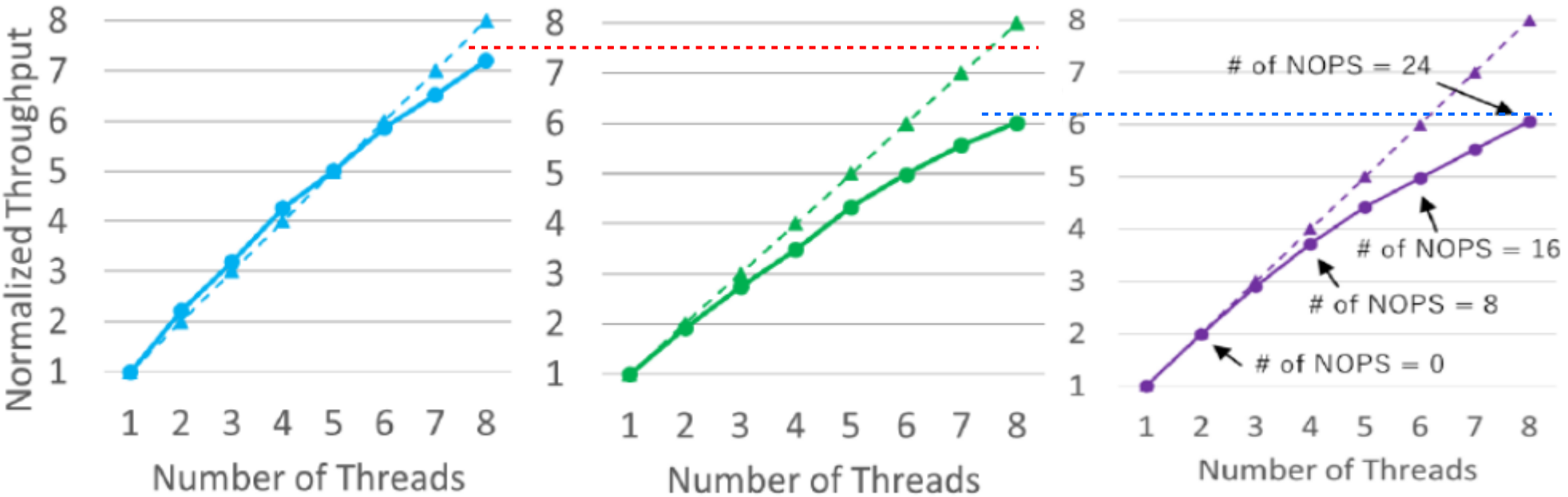
- Dynamic Binary Instrumentation （実行中の命令をいじったり解析したりする）を用いる
  - 利点：言語、アーキテクチャによらない　例えば大規模マシン側が ARM などでも対応できるはず？（未検証）
- QEMU の full emulation mode （遅い）を利用



- QEMU の中間命令上でメモリ読み書き命令の後に NOP を挿入
- 読み込みだけ遅く、なども可能
- 手元のマシン / 大規模マシンどちらの命令セットにも依存しない
- ただし遅いので「バランス」のみしか再現できない

予備評価

- 行列 - ベクトル積のスケーラビリティを比較
  - メモリレイテンシが効くアプリケーション



Machine 1

Machine 2

Machine 1
+ 遅延挿入

性能バランスが異なるマシンでのパフォーマンスを予測可能

Key Takeaways

- マシンの構成だけでなく同一構成でも性能バランスが変化
- 特にプロセッサ flops vs. メモリレイテンシの性能差の開きが大きい
- 手元のマシンと大規模共有マシンで異なる性能バランス
  → 手元で性能の予測とチューニングがしたい
- メモリアクセス命令に遅延を挿入することで性能バランスが異なるマシン上でアプリのパフォーマンスを予測できる

全体のまとめ

- AI センターでシステムソフトウェアを研究する意義
  - マシンの構成の複雑化・性能バランスの変化
  - ユーザから見たシステムと実際のシステムのギャップ

  → ユーザの負担なく性能を最大限引き出すため

- そのためにまず性能の理解が重要
  - 新たなデバイス（不揮発性メモリ）を用いる場合
  - DRAM を低消費電力で利用する場合
  - 異なる性能をバランスのマシン上で実行する場合

- 今後はこれらの知見を活かしたシステムソフトウェアを研究