# 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

東京大学
情報理工学系研究科 創造情報学専攻

○穐山(あきやま) 空道，塩谷 亮太

# 背景

- メインメモリは実行速度・消費電力の両面でコンピュータ性能の大きなボトルネック

- 消費電力
  - AI・ビッグデータ処理の隆盛によるメモリ容量の増加
    - NVIDIA DGX2 では 1.5TB/node

  → マシン消費電力の 25～40% がメモリで消費されるケースも
- 実行速度
  - メモリレイテンシは年代が進んでもほぼ一定

  → flops に対しメモリレイテンシは相対的に増加

# メモリレイテンシ

- メモリレイテンシがほぼ一定な理由
  - レイテンシは並列化で改善不能
  - キャッシュ階層を追加（L2 → L3）するとレイテンシ悪化
  - DRAM 内部の電気的動作のレイテンシはプロセス縮小で改善せず

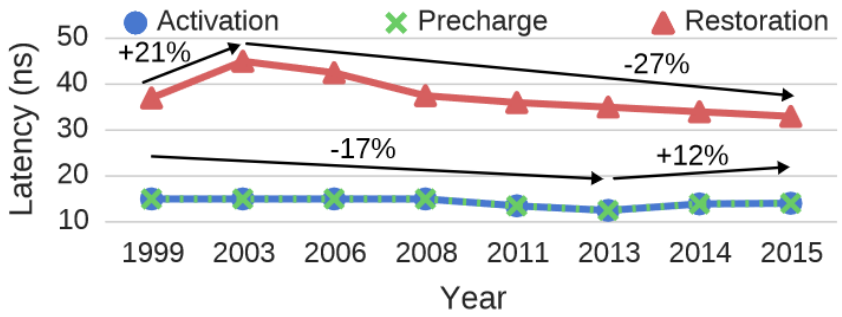



Figure 1: DRAM latency trends over time [20, 21, 23, 51]. (*) より引用

- 一方で CPU コアの性能は指数的に向上

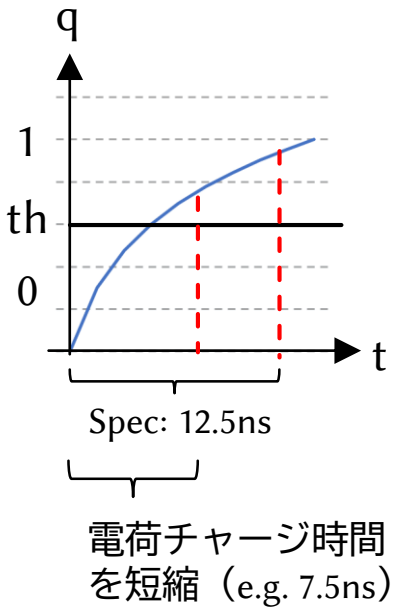
→ CPU コア性能とレイテンシの比は年々増大

(*) K. K. Chang *et al.*, Understanding Latency Variation in Modern DRAM Chips, *SIGMETRICS'16*

# Approximate Memory

- Approximate Memory
  - データにエラーが入ることを許す <-> 低レイテンシ・低消費電力
  - アプリ自体にエラー耐性のある AI、ビッグデータ処理に有用

- 既存研究 (*1, *2, *3 他) の積極的適用で実現可能
  - DRAM 内の電気的動作（電荷の移動）の待ち時間が仕様で規定
  - 待ち時間を仕様より短くし高速・省電力（但し高いエラー率）を実現




(*1) K. K. Chang *et al.*, Understanding Latency Variation in Modern DRAM Chips, *SIGMETRICS'16*
(*2) H. Hassan *et al.*, ChargeCache: Reducing DRAM latency by exploiting row access locality, *HPCA'16*
(*3) X. Zhang *et al.*, Restore truncation for performance improvement in future DRAM system, *HPCA'16*

# Approximate Memory のためのデータ分離

- Approximate Memory をソフトから使うには「データ分離」が必須
  - 保護データ　エラー混入を許さない（バイナリ、ポインタなど）
  - 近似データ　エラー混入を許す（計算用の行列など）

  → 2種類のデータに異なるエラー率を設定

- DRAM 上で「物理的に離れた場所」に配置する必要
  - 現在のメモリシステムの仕組み上、ビット毎のエラー率設定は非効率
  - DRAM セルの row （8K byte） 毎の異なるエラー率設定が現実的
    - 詳細は予稿の 2.2 節と図1

  → 保護データと近似データを異なる row に配置

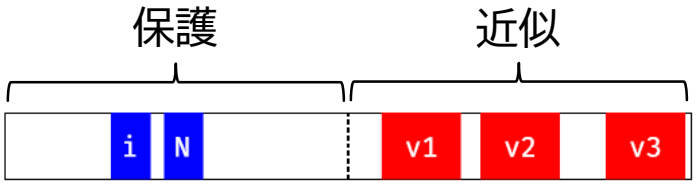
# データ分離による性能低下 (1/2)

- 二種類のデータ分離パターン
  - 疎粒度　大きなメモリ領域（行列など）をそのまま近似データとする
  - 細粒度　構造体内の一部メンバのみを近似データとする

```
int i, N;

double *v1 = malloc_approximate(SIZE);
double *v2 = malloc_approximate(SIZE);
double *v3 = malloc_approximate(SIZE);

for(i=0; i<N; i++) {
    v3[i] = v1[i] * v2[i];
}
```

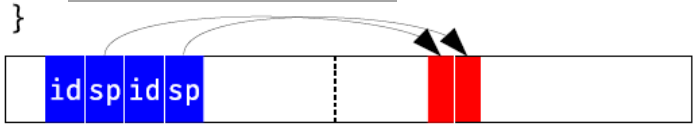



疎粒度

```
typedef struct {
   int id;
   //double s;
   double *sp;
} node;

node *n = malloc(sizeof(node) * N);
double *sps = malloc_approximate(
                    sizeof(double) * N);

for(i=0; i<N; i++) {
    アクセス（次ページ）
}
```




細粒度

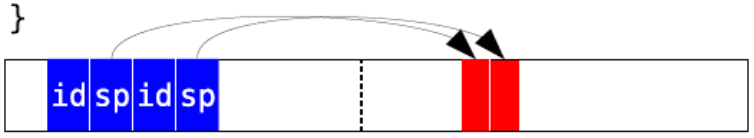
# データ分離による性能低下 (2/2)

- 細粒度データ分離では性能低下の可能性
  - データ分離によるアクセス局所性の悪化
  - 近似データを指すポインタへのアクセスコスト（下図）

```
typedef struct {
   int id;
   //double s;
   double *sp;
} node;

node *n = malloc(sizeof(node) * N);
double *sps = malloc_approximate(
                  sizeof(double) * N);

for(i=0; i<N; i++) {
    n[i].id = i;
    //n[i].s = 0.0;
    n[i].sp = sps[i];
    *(n[i].sp) = 0.0;
}
```

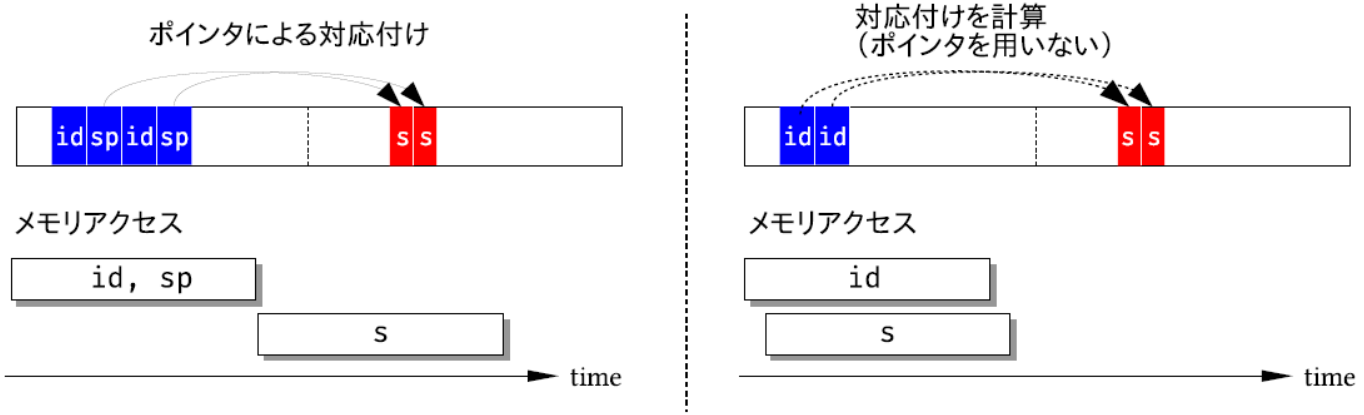



構造体内の double s を近似データにする例

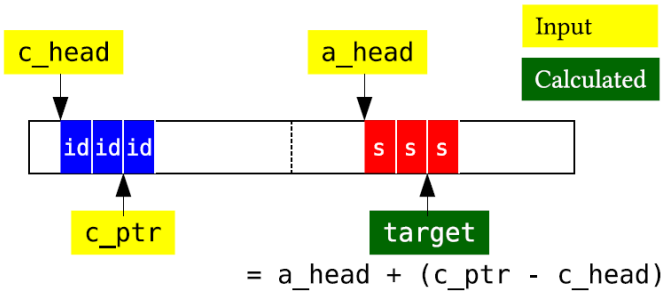
- 対応付けの動作
  - s 用に確保された領域へのポインタを元の構造体に格納
- データアクセス時
  - s の位置を知るために sp が必要
  - sp がロードされるまで s (== *sp) にはアクセス不能

# 提案するアクセス手法

- 基本アイディア
  - 近似・保護データは1対1対応：ポインタなしで位置計算可能
  - 近似・保護データは物理的に遠い：メモリ並列性で同時フェッチ可能




- 近似データの位置計算
  - 先頭から n 番目の保護データにアクセス
    → n 番目の近似データを同時にフェッチ
  - 計算に使う値はキャッシュに載る



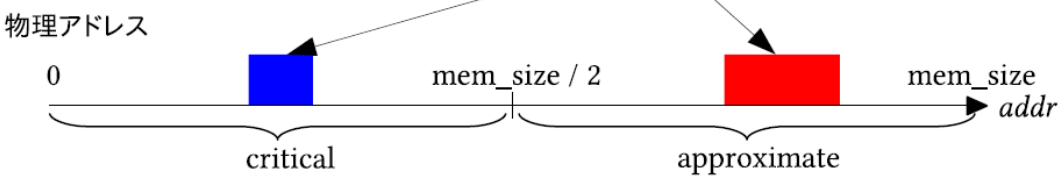
# gem5 での実装 (1/2)

- 保護データと近似データを異なる row に配置するメモリアロケータ
  - malloc_normal, malloc_approximate を提供
  - mmap に MAP_APPOXIMATE フラグを追加し、malloc_approximate から利用

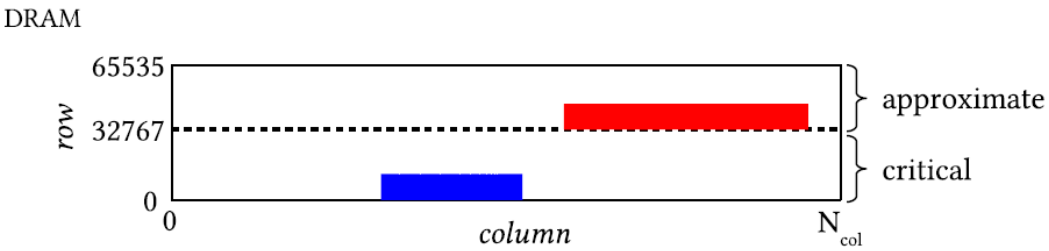
プログラム

```
void *malloc_approximate (size_t size) {
  ...
  mmap(..., flags | MAP_APPROXIMATE, ...);
  ...
}

void *d1 = malloc_approximate(s1);
void *d2 = malloc_normal(s2);
```




保護データと近似データを物理アドレス上で分離する




row は最上位 X ビットで決まるので自動的に異なる row のセットに保持される

# gem5 での実装 (2/2)

- 近似データにアクセスするアクセサ
  - ロードしたい保護データの位置に加え、保護データ全体の先頭位置、近似データ全体の先頭位置、を与える

  → ロードしたい保護データと、対応する近似データを同時にフェッチ

```
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  }
```

  - 任意の型を許すためひとまず C のマクロで実装
    - ユーザから透過的なインターフェースは今後の課題
  - C 言語上は直列だが、*c_ptr と *target は物理的に離れており DRAM 内の並列性で同時にフェッチ可能

# 評価実験

- 評価項目
  1. ポインタを用いたナイーブなデータ分離による性能低下
  2. 提案手法による性能低下の抑制
  3. 提案手法による row activation 数の変化（省略）
- 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 |

- ワークロード
  - gemm: 行列・行列積
  - qsort: 配列のクイックソート
  - merge sort: linked list のマージソート
  - struct seq: 構造体の配列をシーケンシャルアクセス
  - struct rand: 構造体の配列をランダムアクセス

# 評価プログラムのコード例

- 近似データのサイズに対する性能変化を見るため、8×Dバイトのダミーデータを近似データに挿入

データ分離なし

```
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
```

保護データ

近似データ

……

データ分離あり

```
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
```

保護データ

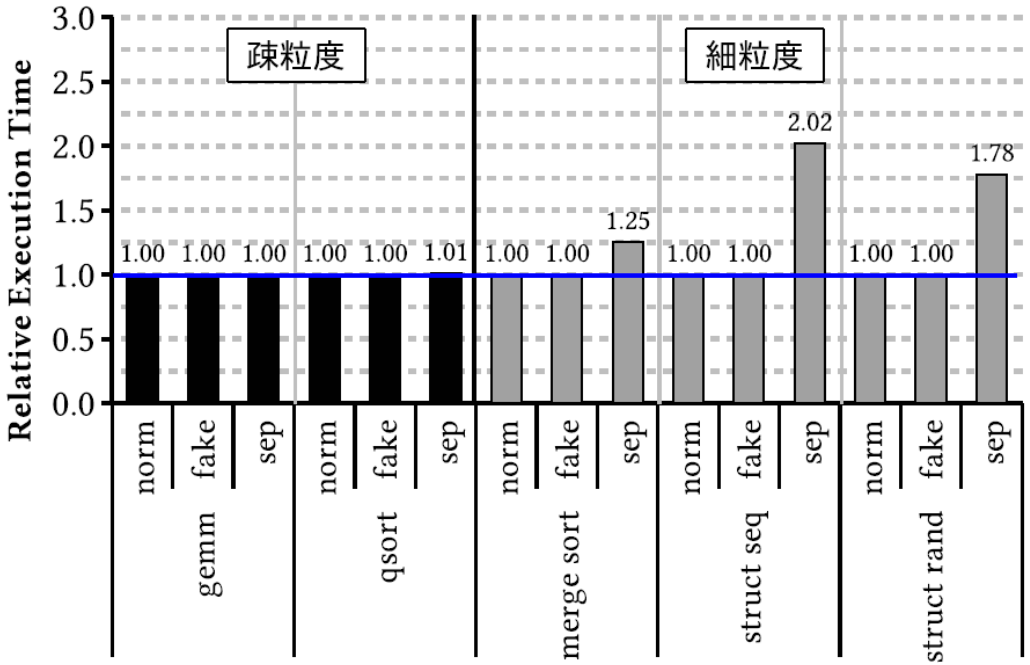
近似データ

……

分離なしの場合のデータのサイズを Sn1
分離ありの場合の保護データのサイズを Sn2
分離ありの場合の近似データのサイズを Sa

} とする。 Sn2 = 8 , Sn1 = Sn2 + Sa

# 1. ポインタを用いたデータ分離による性能低下

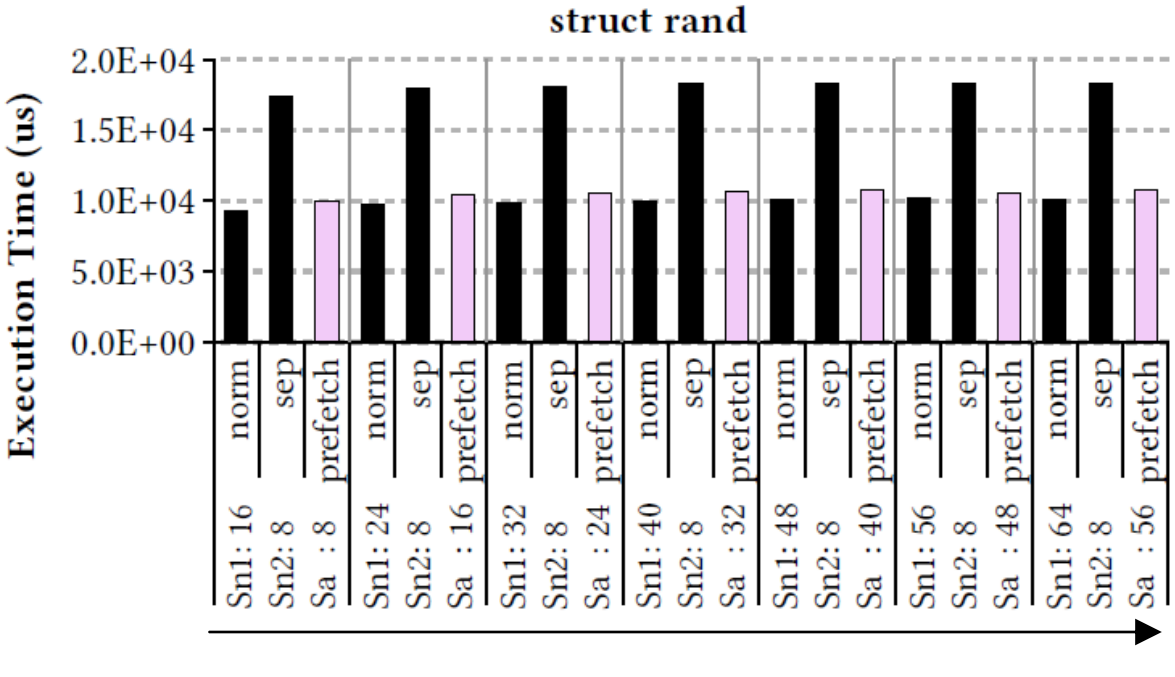



D = 0 のケースで評価
norm: 分離なし
fake: 使う row 数を削減、分離はなし
sep: ポインタを用いた分離

- 疎粒度では性能低下なし
- 細粒度の fake では性能低下なし
- 細粒度の sep では 25～102 % の性能低下
  - データ分離による効果

ナイーブなデータ分離では性能低下が大きい
→ Approximate Memory を活用できない

# 2. 提案手法による性能低下抑制 (1/2)

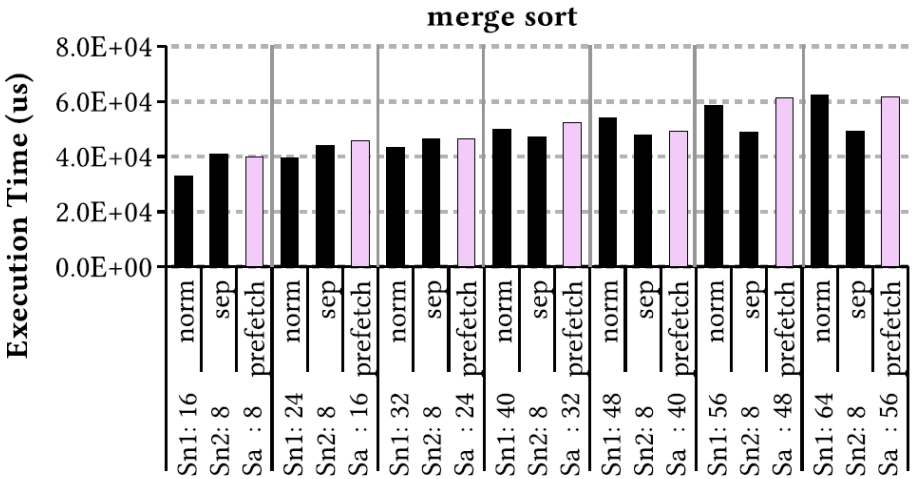


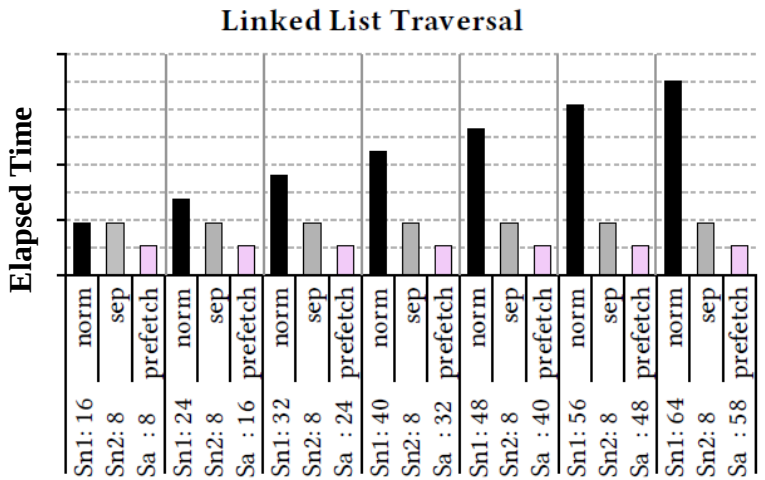

norm: 分離なし
sep: ポインタを用いた分離
prefetch: 提案手法

- 全データサイズで prefetch と norm がほぼ同じ（struct seq も同様のため省略）
- 予稿からデータ更新
  - gem5 で idiv 命令が長大な uops に展開されるため削除
  - partial flag stall による偽の依存を解消

全てのケースで性能低下をほぼ完全に抑制
→ 提案手法により Approximate Memory を活用可能

# 2. 提案手法による性能低下抑制 (2/2)




- 左図：データサイズが大きくなると、ナイーブな分離で性能が*向上*
  - 右図：データ分離すると保護データ（次の node へのポインタ）が小さな領域に集まり、linked list の traversal が大きく高速化
  - 同じことが prefetch にも言えるが、prefetch では性能悪化（原因調査中）

アプリによってはナイーブなデータ分離で性能が向上
→ prefetch でなぜ遅くなっているのか調査が必要

# 関連研究

- デバイスレベルで DRAM を高速・省電力化する研究
  - 製造ブレから生じるエラー耐性の差や、現在の電荷量に合わせて row activation 等の時間を削減する研究が盛ん
  - 多くの場合「ソフトウェアからどう使うか」は議論されず
- Approximate Computing 一般の研究
  - EnerJ (*1, *2): 近似データ用のロード・ストア命令を用意し、プログラミング言語レベルで近似データと保護データの分離を保証

→ 近似データと保護データでキャッシュラインを分けているがその詳細・影響は議論されず

  - 文献 (*3): refresh 頻度を削減したときの化けやすさ（エラー耐性）を row ごとに測り、化けにくい row に保護データを置く

→ 本研究と同様にデータ分離が必要だが、詳細には議論されず

ソフトから使う場合の議論が欠けており、本研究はそこを議論

(*1) H. Esmaeilzadeh *et al.* Architecture Support for Disciplined Approximate Programming, *ASPLOS'16*
(*2) A. Sampson *et al.*, EnerJ: Approximate Data Types for Safe and General Low-power Computation, *PLDI'11*
(*3) A. Raha *et al.*, Quality Configurable Approximate DRAM, *IEEE Transactions on Computers*, Vol. 66, No. 7 (2017)

# まとめと今後の課題

- メインメモリの消費電力とレイテンシを削減するため、Approximate Memory の適用が有力
- しかし保護データと近似データを物理的に分離する必要があるため、細粒度データ分離では性能が低下することがある
- 本発表では gem5 を用いデータ分離パターンやデータサイズが変化する場合の性能低下を定量評価し、性能低下を抑制する手法を提案した

- 今後の課題
  - 性能低下が抑制できていないケース（merge sort）の調査・解決
  - SPEC CPU などのリアルなワークロードでの評価
  - プログラムから透過的に利用できるインターフェスの提案

# 補足：DRAM の内部動作

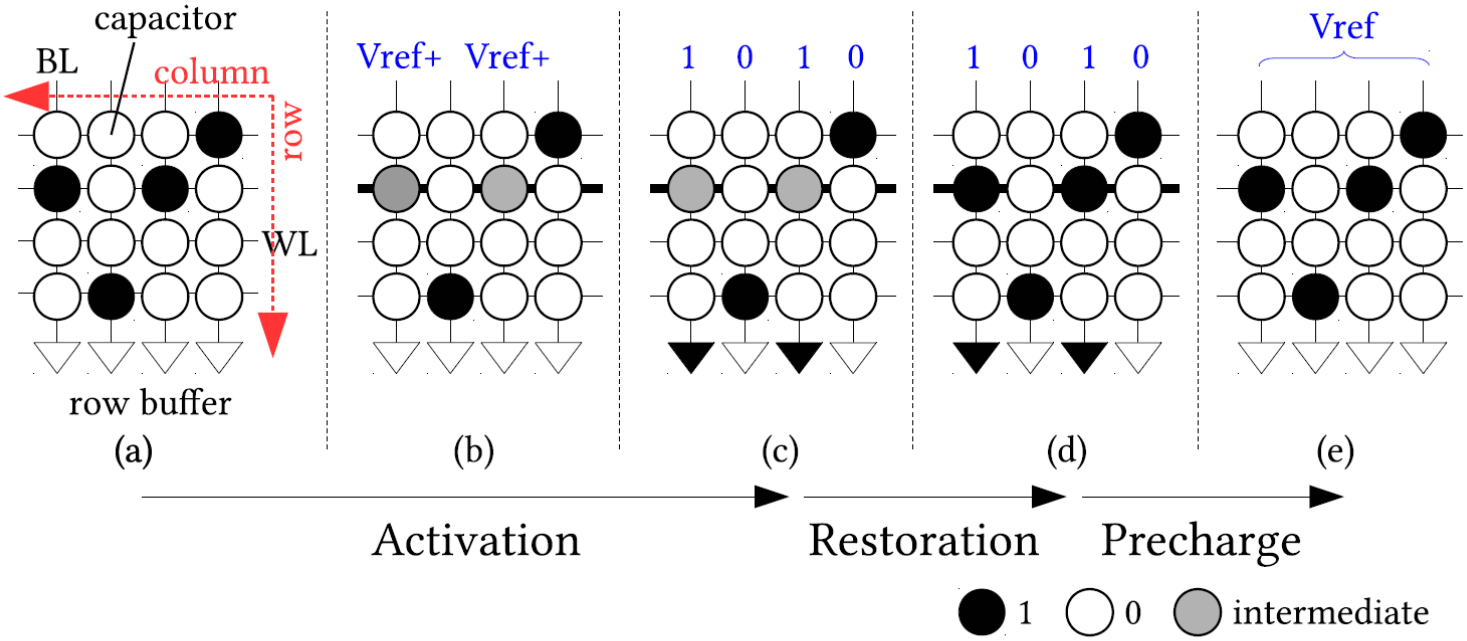



**Fig. 2** DRAM Internal Operations: Activation, Restoration, Precharge

(*) より引用

(*) S. Akiyama, “XXXXXXX”, submitted to *IEICE Transactions on Information and Systems*