図 9 mcf への AoS から SoA への擬似的変換の影響

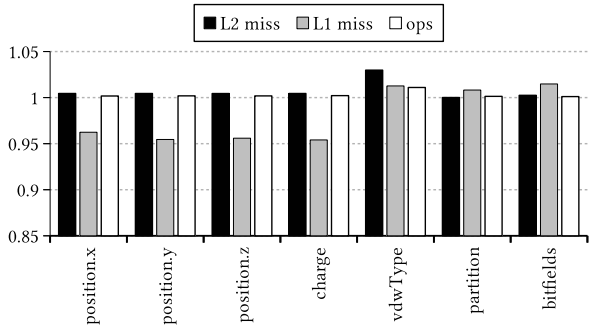

図 10 namd への AoS から SoA への擬似的変換の影響

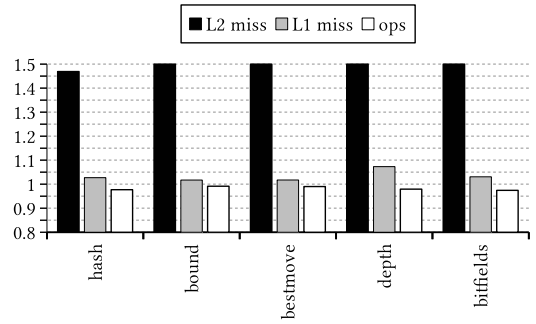

図 11 deepsjeng への AoS から SoA への擬似的変換の影響

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).