不揮発性メモリを用いたJavaオブジェクト永続化のオーバヘッドの調査

松本 康太郎　高田 喜朗　鵜川 始陽

近年，電源が喪失してもデータを保持することが可能な主記憶装置である不揮発性メモリが市場に登場した．この不揮発性メモリを Java などのマネージド言語で自然に利用できるようにするために，オブジェクトが自動的に永続化される仕組を持つ永続化ヒープが開発されている．本研究では，予想されるオブジェクト永続化のためのオーバヘッドを調査した．そのために，OpenJDK に不揮発性メモリへのアクセスを伴う処理を実装した．その結果，全てのオブジェクトを永続化した場合，ベンチマーク毎に最大で 200–500%のオーバヘッドが観測された．

1　はじめに

近年，電源を喪失してもデータを保持することができる主記憶装置の不揮発性メモリ（NVM）が市場に登場した [5]．NVM は通常の DRAM と同様にロード・ストア命令でバイトアクセスが可能であり，書き込まれたデータは電源喪失などによるクラッシュ後も消えることのない永続化された状態となる．この特性を利用して，クラッシュ後にプログラムの状態を復元させることができるシステムを構築できる．

メモリへのアクセスはキャッシュを介して行われるため，書き込みを永続化させるにはキャッシュラインを NVM に書き戻す必要がある．キャッシュラインは自動的にも書き戻されるが，その順序はプログラム上の書き込みの順序と一致するとは限らない．そのため，明示的にキャッシュを書き戻して，NVM 上のデータの整合性を維持しなければ，クラッシュのタイミングによってはプログラムの状態を復元できなくなる．Java のようなマネージド言語ではメモリ管理を行う機構が備わっており，メモリについてプログラマが意識しなくて良い．このようなマネージド言語で NVM を利用するため，キャッシュラインの書き戻しを意識することなく自動的にオブジェクトを永続化する仕組みを持つ永続化ヒープが開発されている．

Shull らは，Java においてユーザが指定した static 変数から参照を辿ることで到達可能なオブジェクトの最新の状態が自動的に永続化されるというプログラミングモデルを提案し [6]，その実装を示した [7]．しかし，オブジェクトを永続化しない場合に比べて，どの程度遅くなるかは示されていない．

我々は，Shull らのプログラミングモデルを実現する新しいアルゴリズムを開発している．本研究では，我々のアルゴリズムのうち NVM へのアクセスを伴う処理のオーバヘッドやその原因の調査を行う．主なオーバヘッドが NVM への書き込みに由来するとすれば，Shull らの実装 [7] と同程度のオーバヘッドになると考えられる．しかし，Shull ら [6] [7] はオブジェクト永続化の仕組みがない VM（仮想機械）との，NVM の実機を用いた比較を示していないので，本研究で調査する．そのために，OpenJDK に NVM へのアクセスを伴う処理を実装する．ただし，実装した VM は x86 系 CPU 上で動作することを前提とする．

An Overhead Evaluation for Persistent Java Objects Using Non-Volatile Memory

Kotaro Matsumoto, Yoshiaki Takata, Tomoharu Ugawa, 高知工科大学情報学群, School of Information, Kochi University of Technology.

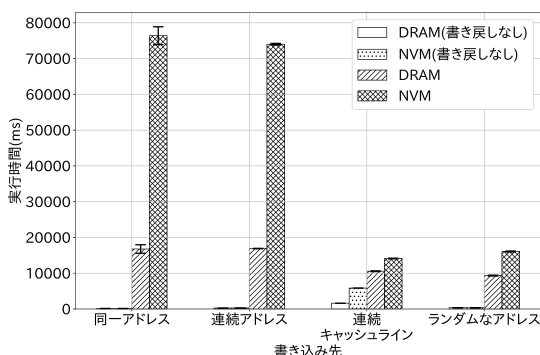

図 1　DRAM と NVM の書き込みにかかった時間

2　不揮発性メモリ

2.1　キャッシュとメモリバリア

NVM への書き込みは，キャッシュを介して行われる．マシンが電源を喪失するとキャッシュ上のデータは消えてしまうため，書き込みを行ったキャッシュラインが NVM に書き戻されて初めて，その書き込みが永続化される．永続化されたデータの整合性を維持するには，キャッシュラインを強制的に書き戻す命令を使って，キャッシュラインを書き戻す順序を制御する必要がある．本研究では，キャッシュ上のデータを無効化せずにキャッシュラインの書き戻しを行う clwb（Cache Line Write Back）命令を用いる．

clwb 命令は後続の書き込み命令や clwb 命令と並行して動作する可能性があるので，clwb 命令だけでは意図しない順序で書き込みの永続化が行われる可能性がある．キャッシュラインの書き戻しの順序を保証するには，後続の順序を付けたい書き込みの前にメモリバリアである sfence 命令を挿入し，clwb 命令の完了を待つ必要がある．

2.2　書き込み速度に関する予備調査

1 バイトの書き込みを行う処理を 1 億回繰り返す C 言語プログラムの実行時間を計測した．書き込み先は，DRAM または NVM 上の，同一アドレス，連続アドレス，連続キャッシュライン（64 バイト間隔のアドレス），ランダムなアドレスの 4 種類で実験し，書き込むアドレスによる性能の変化があるか調べた．

図 1 に結果を示す．clwb 命令と sfence 命令によってキャッシュの書き戻しを行うと，DRAM でも NVM でも遅くなった．特に，NVM 上の同一アドレスや連続アドレスに書き込む場合，キャッシュの書き戻しを行わない場合に比べて，それぞれ 588 倍と 262 倍の時間がかかった．このことから，同一キャッシュラインの書き戻しが連続すると遅くなることが分かる．

3　調査対象の VM

3.1　Shull らのプログラミングモデル

Shull らのプログラミングモデル[6] では，ユーザが専用のアノテーションで指定した static 変数を開始点として，そこから参照を辿ることで到達可能な全てのオブジェクトが自動的に永続化される．永続化の対象であるオブジェクトへの書き込みはプログラムと同じ順序で永続化される．これにより，ある書き込みが永続化されていれば，プログラム順序でその書き込みより前にある書き込みや，その書き込みが依存している他のスレッドの書き込みが永続化していることを保証している．

3.2　我々のアルゴリズム

我々のアルゴリズムでは，全てのオブジェクトは DRAM 上に存在し，Java のプログラムは DRAM 上のオブジェクトを使って実行する．永続化対象になったオブジェクトは，コピーを NVM 上に作成する．永続化されたオブジェクトへの書き込みは，書き込みバリアによって DRAM と NVM の両方に行い，読み込みは DRAM からのみ行う．

3.3　オーバヘッド調査のための実装

本研究では，全てのオブジェクトを永続化の対象とした時の，ヒープの永続化，つまり，NVM 上のコピーの作成と維持のオーバヘッドを調べる．この実験では，DRAM 上に新しいオブジェクトが生成される際に，NVM 上にコピーを作成する．NVM 上のメモリは，VM 初期化時に十分に確保しておいた NVM 上の連続したメモリプールを端から切り取って割り当てる．一度割り当てたメモリは解放しない．この時，

ロックを用いて排他制御する．

実装は，JDK のオープンソース実装である OpenJDK 13 を変更することで実現した．JIT（Just-In-Time）コンパイラは使用せず，常に Template Interpreter で処理することを前提とする．本研究では，NVM への書き込みは排他制御していない．そのため，NVM 上のデータは壊れている可能性がある．しかし，Java プログラムは DRAM 上のデータのみを使って実行するので計測は可能である．排他制御のオーバヘッドの計測は今後の課題である．

4 実験

4.1 システム構成

実験に用いたシステムの構成を示す．

- CPU: Intel Xeon Gold 6240（2.60GHz）
- DRAM: DDR4 2666MHz
- NVM: Intel Optane DC Persistent Memory
- OS: Ubuntu 18.04.4 LTS
- C Compiler: gcc version 7.5.0

4.2 使用した VM

変更前のオリジナル VM と，コピーの作成先となるメモリや明示的なキャッシュ書き戻しの有無が異なる VM の合計 4 種類を実装し，実験を行った．

- オリジナル VM
- NVM コピー VM（書き戻しなし）
- DRAM コピー VM
- NVM コピー VM

4.3 書き込み頻度の違いによる実行時間への影響

Java プログラムで書き込み頻度の違いによって実行時間がどのように変化するか調べるため，同じオブジェクトのフィールドに対して 1 億回の読み書きを行う Java ベンチマークを作成した．書き込みの割合を 0%，1%，5%，10%，25%，50%，75%，100%と変化させた時の実行時間が，図 2 である．

結果より，明示的なキャッシュラインの書き戻しがない場合，オリジナル VM との差は最大で 1.3 倍であり，書き戻し以外のオーバヘッドは小さかった．明示的なキャッシュの書き戻しがある場合，NVM コピー

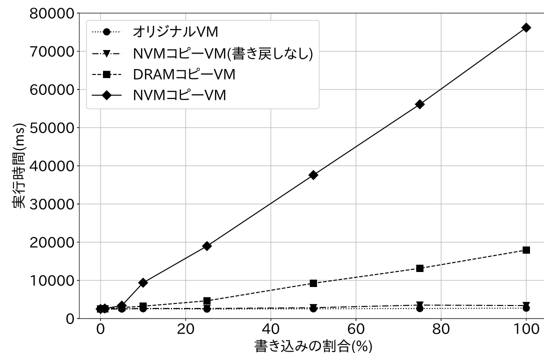

図 2 書き込みの割合を変化させたときの実行時間

VM の実行速度は書き込み 100%でオリジナル VM の 28 倍になった．一方で，書き込みの割合が減少するにつれて実行速度が一定の割合で速くなった．ただし，書き込みが 10%以下のときはオリジナルの VM とほぼ同程度になった．

4.4 DaCapo ベンチマーク

実用的なプログラムでのオーバヘッドを調査するため，DaCapo Benchmark Suite [1] を用いて実行時間を計測した．DaCapo Benchmark Suite は，複雑なメモリアクセスを伴う実際のアプリケーションで構成されているベンチマークスイートである．OpenJDK 13 で実行できない †1 batik，eclipse，tomcat を除く全てのベンチマークの実行時間を，各々の VM で計測した．

図 3 は，オリジナル VM の実行時間を 1 に正規化したときの比を表している．明示的なキャッシュの書き戻しがない場合，オリジナル VM と比較すると最大で 1.3 倍であり，書き戻し以外のオーバヘッドは小さかった．明示的なキャッシュの書き戻しがある場合，多くのベンチマークでおおよそ 200%から 500%のオーバヘッドが観測された．h2，tradebeans，tradesoap の 3 種類はオーバヘッドがおおよそ 100%以下と，他のベンチマークより比較的小さくなった．

†1 DaCapo Benchmark Mailing Lists (https://sourceforge.net/p/dacapobench/mailman/message/36013609/), JDK Bug System (https://bugs.openjdk.java.net/browse/JDK-8155588)

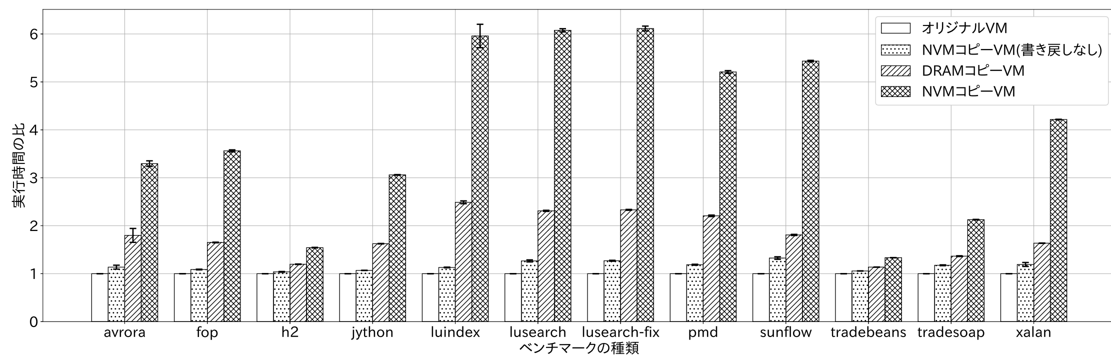

図 3　**DaCapo** ベンチマークの実行時間

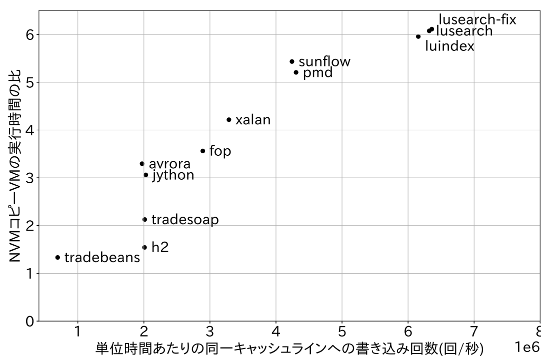

図 4　単位時間あたりの同一キャッシュラインに対する連続書き込み回数と実行時間

4.5　オーバヘッドの原因

4.4 節でプログラムによってオーバヘッドの大きさが異なる原因を調査した．その結果，2.2 節で示した同一キャッシュラインへの書き込みが主な原因であることが分かった．

各プログラムの実行において，連続で同一キャッシュラインに書き込みを行った回数を，図 4 に示す．横軸を単位時間あたりの同一キャッシュライン書き込み回数，縦軸はオリジナル VM に対する NVM コピー VM の実行時間の比を表している．

4.6　考察

Java においても書き戻しによる実行時間への影響は大きく，特に同一キャッシュラインへの連続した書き込みは大きなオーバヘッドを伴う．同一キャッシュラインに対する単純な書き込みを行う Java ベンチマークでは，最大で 2700%のオーバヘッドが観測された．一方で，DaCapo ベンチマークは最大で 500%程度のオーバヘッドであった．DaCapo ベンチマークのように実用的なプログラムでは，ヒープへの書き込み以外の処理や異なるキャッシュラインへの書き込みが多く含まれており，連続で同一キャッシュラインに何度も書き込むことは頻繁にないため，プログラム全体の実行速度に与える影響は少ない．

本実験では全てのオブジェクトを永続化の対象としたが，我々が採用している Shull らのプログラミングモデルでは，ユーザによって間接的に指定されたオブジェクトのみが永続化の対象となる．その場合，単位時間あたりの NVM への書き込みはさらに減るため，オーバヘッドはさらに小さくなると期待できる．

5　関連研究

Shull らは，マネージド言語で NVM を使えるようにするためのプログラミングモデルを提案した[6]．また，DRAM を用いることで NVM へのアクセスを伴う処理以外のオーバヘッドを計測している．本研究では，このモデルを実現するためのオーバヘッドを，NVM の実機を用いて調査した．

C++向けの永続化ヒープには，NV-Heaps[3] や NV-HTM[2]，MOD[4] などがある．NV-Heaps や NV-HTM はプログラムにディレイを挿入することで NVM をエミュレートして，MOD は NVM の実機を用いて性能を評価している．これらのライブラリは，

オブジェクトのアクセスにログベースのトランザクションを用いるため，我々のアルゴリズムより NVM への書き込みが多くなるが，キャッシュラインの書き戻しを効率的に行える可能性がある．我々はオブジェクトへの個々の書き込みを直ちに永続化するアルゴリズムのオーバヘッドを調査した．

Java 用の永続化ヒープを実現した研究には，Espresso [8] や AutoPersist [7] がある．AutoPersist は，Shull らが提案したプログラミングモデル [6] を Shull ら自身が Maxine JVM 上に実装したものである．NVM の実機を使って Espresso と比較しているが，オブジェクトを永続化しない場合との比較は示されていない．Espresso は NVM に Viking NVDIMM を用いてデータを永続化した場合と，NVM の代わりに DRAM を用いた場合の性能を比較している．しかし，Espresso もトランザクションを用いてオブジェクトにアクセスするプログラミングモデルを採用しており，トランザクションや永続化に関連する処理を一切含まない処理系との比較は行われていない．

6 おわりに

NVM をマネージド言語で使うためには，キャッシュの書き戻しを意識することなくオブジェクトを永続化できる仕組みが必要となる．本研究では，オブジェクトの永続化に必要な処理のうち NVM へのアクセスを伴う部分を OpenJDK に実装し，全オブジェクトの永続化にかかるオーバヘッドを調査した．実験の結果，単純な書き込みを行うベンチマークでは最大で 2700%のオーバヘッドが観測された．しかし，全アクセスに占める書き込みアクセスの割合が 10%以下になると，NVM の代わりに DRAM を使った場合と同程度のオーバヘッドになった．DaCapo ベンチマークで観測されたオーバヘッドは 200%から 500%であり，同一キャッシュラインへの連続アクセスが多いプログラムは遅くなる傾向があった．

参 考 文 献

[1] Blackburn, S. M. et al.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, 2006, pp. 169–190.

[2] Castro, D., Romano, P., and Barreto, J.: Hardware Transactional Memory Meets Memory Persistency, *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*, 2018, pp. 368–377.

[3] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S.: NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, 2011, pp. 105–118.

[4] Haria, S., Hill, M. D., and Swift, M. M.: MOD: Minimally Ordered Durable Datastructures for Persistent Memory, *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020, pp. 775–788.

[5] Intel: インテル®Optane™ パーシステント・メモリー, https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/optane-dc-persistent-memory.html. (2020 年 7 月 18 日閲覧).

[6] Shull, T., Huang, J., and Torrellas, J.: Defining a High-Level Programming Model for Emerging NVRAM Technologies, *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang'18)*, 2018.

[7] Shull, T., Huang, J., and Torrellas, J.: AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, 2019, pp. 316–332.

[8] Wu, M., Zhao, Z., Li, H., Li, H., Chen, H., Zang, B., and Guan, H.: Espresso: Brewing Java For More Non-Volatility with Non-Volatile Memory, *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, 2018, pp. 70–83.