# 不揮発性メモリを用いた複製に基づく Java オブジェクトの永続化

1255119　松本 康太郎　【 ソフトウェア検証・解析学研究室 】

# Replication-Based Java Object Persistence Using Non-Volatile Memory

1255119　MATSUMOTO, Kotaro　【 Software Verification and Analysis Lab. 】

## 1　はじめに

不揮発性メモリ（Non-volatile memory，NVM）は，電源喪失などのクラッシュ後もデータを保持し続けることができる主記憶装置である．NVM は，従来の揮発性メモリである DRAM と同様にロード・ストア命令でアクセスが可能であり，書き込まれたデータはクラッシュ後にも消えない「永続化」された状態になる．

NVM の登場によって，HDD や SSD などの補助記憶装置よりも高速にデータを永続化することが可能になった．しかし，DRAM に比べるとアクセスが低速であるため，必要なデータのみを選択的に永続化するアプローチが有効である．このとき，一部のデータは永続化されず DRAM のみに存在するため，NVM 上に DRAM 上のアドレスを指すポインタを書き込むと，クラッシュ後に dangling pointer となる点に注意しなければならない．また，メモリへのアクセスは揮発性の CPU キャッシュを介して行われるため，キャッシュラインを明示的に書き戻して NVM 上のデータの整合性を保つ必要がある．このように NVM を正しく扱いながら必要なデータを選択的に永続化するのは煩雑であるため，自動的かつ選択的にデータを永続化し，そのデータの完全性を保証する研究が行われている．

到達可能性に基づく永続化は，プログラマが指定した変数（永続ルート）を開始点として，そこから参照を辿ることで到達可能なオブジェクトを永続化する考え方である．先行研究では，Shull ら [1] がこれを実現する Java 仮想機械（VM）の実装を示した．これは，DRAM 上にオブジェクトを作成し，永続化の対象になった時に NVM 上にオブジェクトを**移動**させる．しかし，アクセスの度にオブジェクトが移動しているかどうかを調べる必要がある点と，DRAM よりもアクセスが低速な NVM からオブジェクトの値を読み出す点で，非効率である．

本研究では，先行研究の非効率な点を改善した新しいアルゴリズム Replication Based Persistency（RBP）を提案する．また，クラッシュ後のプログラムの再起動において，NVM 上の複製を用いたデータを復元するリカバリ機構も実現する．この提案アルゴリズムを OpenJDK 16 に実装して，性能を評価する．

## 2　提案手法

### 永続化機構

RBP は，永続化対象となった DRAM 上のオブジェクト（dram copy）の**複製**（nvm copy）を NVM 上に作成することで永続化を実現する．複製を作った後も dram copy は，無効化せずに使い続ける．複製を持つオブジェクト（永続オブジェクト）にも dram copy が存在するため，読み出しは常に DRAM から行うことが可能であり，オブジェクトが移動しているかどうか調べる必要がない．一方，nvm copy も最新の状態に保つため，書き込みは dram copy と nvm copy の両方に行う．このとき書き込む値が参照である場合は，先にその参照から到達可能な全てのオブジェクトの複製を NVM 上に再帰的に作成する．書き込み後は，キャッシュラインを書き戻す `CLWB` 命令と，それを順序付けする `SFENCE` 命令を発行し，キャッシュだけではなく NVM にもデータが書き込まれていることを保証する．

nvm copy 領域は，NVM 専用のメモリアロケータによって確保する．さらに，NVM 専用のごみ集め（GC，ガベージコレクション）の機能も提供する．

### 競合の解決

マルチスレッド環境では，複数のスレッドによる書き込みを永続化する順序が復元した状態に影響する．RBP では，書き込みを Java 言語のメモリモデルで定義された半順序である happens-before order [2] に反しない順序で永続化する．RBP は，プログラムの複数のスレッドが同じオブジェクトに対して同時にアクセスすることはないという仮定の下で，これを実現している．ただし，`java.util.concurrent.atomic` パッケージによるアトミックな読み書き操作と，`volatile` フィールド（`volatile` 修飾子が付いたフィールド）への読み書き操作は，同時にアクセスすることを目的とした機能であるため，同時にアクセスしても happens-before order に反しない順序で永続化する．

これらを実現するために，マルチスレッド環境で RBP を動作させると発生する 4 つの競合に対処する．

**競合 1:** 複数のスレッドが同じオブジェクトの複製を

作成する場合，複製が複数個作成される可能性がある．これを防ぐために，CPU のアトミックな *Compare-and-Swap*（CAS）命令を用いて，複製を作成する担当スレッドを決める．それ以外の複製を作成しようとするスレッドは，担当スレッドを待つ．RBP は，永続オブジェクトから到達可能な複数のオブジェクトを同時に永続化するため，担当スレッドを待つ間に他の複製を作成する．

**競合 2:** オブジェクトへの書き込み中に，別のスレッドがそのオブジェクトの複製を作成した場合，書き込みを行うスレッドが複製が作られたことに気付かず，dram copy だけに書き込みを行う可能性がある．その結果，dram copy と nvm copy は異なる値を持つことになる．これを防ぐために，書き込み処理は先に dram copy に書き込み，次にオブジェクトが永続化されているか調べて，永続化されていれば nvm copy にも書き込む.

**競合 3:** オブジェクトの複製を作成している間に，別のスレッドがそのオブジェクトに対して書き込みを行った場合，書き込み処理で nvm copy に書き込んだ新しい値を，複製の作成処理が古い値で上書きしてしまう可能性がある．その結果，dram copy と nvm copy は異なる値を持つことになる．これを防ぐために，複製の作成処理は自身が nvm copy に古い値を上書きしたことを検出し，新しい値を書き込み直す．具体的には，dram copy から nvm copy に内容をコピーした後に，それらの内容を比較して，異なる場合はコピーをやり直す.

**競合 4:** アトミック操作と `volatile` フィールドの読み書きでは，複数のスレッドが同じオブジェクトに対して同時にアクセスするが，この競合にも対処している．

### リカバリ機構

RBP は，NVM 上のデータを元に，DRAM 上にある NVM 管理情報と dram copy を再構築する機構を提供する．これを実現するために，リカバリに必要なクラス名と永続ルートも永続化する．オブジェクトのリカバリ処理は，復元領域（dram copy）を確保して，そこに NVM 上のデータをコピーする．このとき，過渡的な状態が他の Java スレッドに観測されたり，GC による影響を受けたりしないことを目標とする．この目標は，他のスレッドを全て停止させることで達成できるが，GC が必要な dram copy の確保では，GC スレッドを停止させることができない．そこで，リカバリ処理を大きく‘領域確保のフェーズ’と‘内容コピーのフェーズ’の二つに分割する．つまり，初めに**全ての** dram copy 領域を確保し，その後に各オブジェクトの内容をコピーする．前者は GC を動作させるため，他のスレッドを停止せずに処理する．このとき，確保した dram copy 領域は GC に回収されないよう，Java のローカル変数から参照しておく．後者は他の全てのスレッドを停止させて処理する．これにより，先述の目標を達成できる．

さらに，リカバリ処理中にクラッシュが発生しても問題にならないよう対策した．

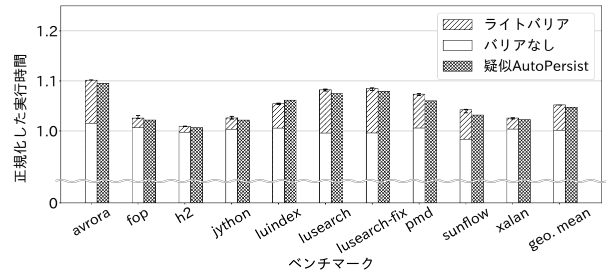


図 1 基本的なオーバヘッド

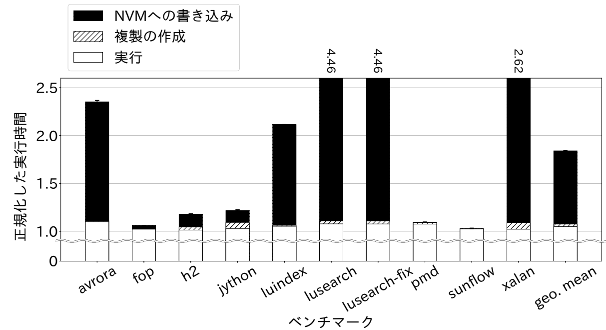


図 2 永続化オーバヘッド

## 3 性能評価

RBP の性能を評価するために，DaCapo Benchmark Suite を用いた実験を行った．

RBP はオブジェクトを選択的に永続化できるため，まず，永続化しなくてもかかるオーバヘッドを調査した．図 1 は，その実験結果である．縦軸は，変更を加えていない元の VM を 1 に正規化した実行時間の比である．オーバヘッドは平均 5.2%であり，先行研究のオーバヘッドの一部を再現した疑似 AutoPersist も同程度のオーバヘッドであった．

図 2 は，全ての static フィールドを永続ルートに設定したときの実験結果である．複製を作成するオーバヘッドは小さく，最大の領域を永続化した xalan でも約 6.9%であった．主なオーバヘッドは NVM への書き込みであり，書き込み頻度が高いベンチマークでは 100%を超えた．読み出しのオーバヘッドは観測されなかった．

また，文字列を生成して永続化するプログラムで，文字列の個数を変化させてリカバリの時間を測定した．リカバリには 1GB あたり約 4.72 秒の時間が必要であった．

## 4 おわりに

本研究では，到達可能性に基づく永続化を実現する新しいアルゴリズムを提案した．また，これを OpenJDK 16 に実装し，オーバヘッドとその内訳を調査した．

## 参考文献

[1] T. Shull, J. Huang, and J. Torrellas. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability. In *PLDI’19*, pages 316–332, 2019.

[2] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL’05*, pages 378–391, 2005.