令和 4 年度

修士学位論文

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

Replication-Based Java Object Persistence Using Non-Volatile Memory

1255119　　松本 康太郎

指導教員　　高田 喜朗

2023 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

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

松本 康太郎

不揮発性メモリ（Non-volatile memory，NVM）は，電源を喪失してもデータを保持し続ける主記憶装置である．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．このアプローチでは，一部のデータは永続化されず DRAM のみに存在するため，NVM 上に永続化されていないデータを指すポインタを書き込むと，クラッシュ後に dangling pointer となり，リカバリできない．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もある．

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り，これをプログラム中に明示的に記述することは，プログラマの大きな負担となる．一方，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．そこで，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．先行研究では，Shull らが到達可能性による永続化を実現する AutoPersist と呼ばれる Java 用の永続化ヒープの実装を示した．到達可能性に基づく永続化とは，プログラマがアノテーションを用いて指定した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オブジェクトを永続化対象とする考え方である．AutoPersist は，DRAM 上にオブジェクトを作成し，オブジェクトが永続化の対象になると NVM 上にオブジェクトを移動させる．この実装は，読み出し時にオブジェクトが永続化

対象であるか確認する処理が必要である点と，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，到達可能性による永続化を実現する新しいアルゴリズムを提案する．先行研究と異なり，読み出しは常に DRAM から行うことが可能であり，読み出し時の追加処理を必要としない．オブジェクトの永続化は，オブジェクトを NVM 上に移動させるのではなく，NVM 上にオブジェクトの複製を作成することで実現する．複製の作成後は，DRAM 上のオブジェクトと NVM 上のオブジェクトの内容を同期する．これがマルチスレッド環境でも正しく動作するよう，考えられる全ての競合を解決した．ただし，書き込みでは基本的にロックを用いた排他制御を行わず，細粒度の同期処理によって競合を解決するよう設計した．このような永続化機構に加えて，電源喪失などのクラッシュ後のプログラムの再起動において，NVM 上の複製を用いてデータを復元するリカバリ機構も実現する．具体的には，DRAM 上のヒープにオブジェクトの領域を確保し，その内容を複製から書き戻す．このとき，システムのクラッシュやごみ集めが発生しても問題にならないよう設計した．

提案アルゴリズムを実用的な Java 仮想機械である OpenJDK 16 に実装して，オーバヘッドを評価した．実験結果より，複製を作成するオーバヘッドは小さく，読み出しのオーバヘッドはほぼ存在しないことがわかった．また，永続化したオブジェクトへの書き込みのオーバヘッドは，ほとんどが NVM への書き込みによるものであった．オブジェクトを永続化しないプログラムのオーバヘッドは，先行研究である AutoPersist と同程度であることがわかった．また，実際に提案システム上でプログラム実行中に電源を切断し，正しく復元できることを確認した．これらの機構を用いる Java アプリケーションの作成手順を示し，リカバリに必要な時間の傾向も調査した．

キーワード　不揮発性メモリ，到達可能性，オブジェクト永続化，並行性，Java

Abstract

Replication-Based Java Object Persistence Using Non-Volatile Memory

MATSUMOTO, Kotaro

Non-volatile memory (NVM) is main memory that retains data even when the power of the machine is lost. The emergence of NVM has made it possible to persist data faster than secondary storage devices such as HDD and SSD. However, the access speed of NVM is slower than DRAM. Therefore, an approach to selectively persist the necessary data is effective. Because, in this approach, some data is not persisted and exists only in DRAM, writing an address of data that is not persisted to NVM will make it a dangling pointer after a crash, which makes recovery impossible. In addition, because memory access is performed via the volatile CPU cache, the application must explicitly write back cache lines to maintain the integrity of the data in NVM.

Because of the requirements of the special treatment as described above, a program that selectively persists necessary data while correctly handling NVM is complicated. For programmers, to explicitly describe such a program is a burden. On the other hand, managed languages such as Java have a mechanism for memory management and do not require programmers to be aware of memory. Therefore, there are studies to persist data automatically and selectively, guaranteeing its integrity. Shull et al. presented an implementation of a persistent heap for Java called AutoPersist that achieves persistence by reachability. Persistence by reachability is the concept that makes objects reachable by following a chain of references from the variables annotated by the programmer.

AutoPersist creates objects in DRAM and moves them to NVM when they become persistent. This implementation has two inefficiencies. One is the read barriers to get the current location of objects; the other is to read values of persistent objects from NVM, which is slower than DRAM.

This paper proposes a new algorithm to realize persistence by reachability. Unlike AutoPersist, it is always possible to read object contents from DRAM without additional processing. The proposed algorithm does not move objects to NVM to make them persistent. Rather, it creates replicas of the objects in NVM. After replication, the original object in DRAM is kept synchronized with the replica. We resolved all possible race conditions for our algorithm to work correctly in a multi-threaded environment. With a few exceptions, the algorithm uses fine-grained synchronization rather than using locks to resolve race conditions. In addition to this persistence mechanism, we propose a recovery mechanism that recovers data using replicas in NVM after a crash. Specifically, it first allocates areas in DRAM for objects, and then it copies the contents of them from the replicas in NVM. Our designed system works correctly if garbage collection takes places or even the system crashes during recovery.

We implemented it in OpenJDK 16, which is a product-level Java virtual machine , and evaluated its overhead. The results showed that the overhead of making objects persistent was small, and the overhead of reading from objects was negligible. The overhead of writing to persistent objects was mostly due to writing to NVM. The overhead imposed on programs that do not make any object persistent was similar to AutoPersist. In addition, we confirmed that the system correctly recovered the objects after we disconnected the power supply while the program was running under the proposed system and restarted the system.

key words Non-volatile memory, reachability, object persistence, concurrency, Java

目次

図目次

表目次

第 1 章

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 [1]．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ることなく，データを保持し続ける．この NVM に書き込んだデータがクラッシュ後にリカバリ可能であるとき，そのデータは “永続化されている” という．

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 のみに存在している．この永続化されていないデータを指すポインタを NVM に書き込むと，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 CPU キャッシュを介して行われるため，書き込み命令を実行しても実際には CPU キャッシュにだけ書き込まれていて NVM には書き込まれていないような過渡的な状態がある．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．

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り，これをプログラム中に明示的に記述することは，プログラマの大きな負担となる．一方，Java [2, 3] 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．この機構を拡張し，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．

Orthogonal persistency [4] は，自動的かつ透過的にデータを永続化するプログラミング言語の実行環境の機能である．近年，Orthogonal persistency の永続化ストレージとして，HDD や SSD の代わりに NVM を用いる研究が行われている．Orthogonal persistency では，一般的に到達可能性に基づいて永続化するオブジェクトを決定することが多い [5]．到達可能性に基づく永続化とは，プログラマがアノテーションを用いて指定した変数（これを永続ルートと呼ぶ）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対象とする考え方である．

AutoPersist [6] は，オブジェクトの到達可能性に基づいて Java VM で Orthogonal persistency を実現した研究である．AutoPersist は，永続化対象となったオブジェクトを DRAM から NVM に**移動**させることで永続化を実現する．つまり，DRAM 上のオブジェクトを無効化して，以後の読み書きは NVM 上のオブジェクトに対して行う．この実装には，非効率な点が二つある．

一つは，読み出し時にオブジェクトが永続化対象であるか確認する処理が必要な点である．AutoPersist はオブジェクトを NVM 上に移動させた後，移動前の DRAM 上のオブジェクトに移動先を指す参照である転送ポインタを残す．転送ポインタがセットされているオブジェクトへの読み書きは操作は，転送ポインタを辿って NVM 上のオブジェクトに対して行う．参照値の比較処理においても，転送ポインタを考慮した追加の処理が必要となる．オブジェクトを全く永続化しないアプリケーションプログラムにおいても，この処理のオーバヘッドがかかる．

もう一つは，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ある．NVM は DRAM に近いアクセス速度を有しているが，DRAM よりは低速である．Friedman ら [7] は，代表的な NVM の製品である Intel Optane DC Persistent Memory の読み出しスループットが DRAM と比べておよそ 1/3 であることを示した．

本研究では，AutoPersist の二つの非効率な点を改善するアルゴリズム Replication Based Persistency（以後，RBP と呼ぶ）を提案する．RBP は，書き込み操作の結果を

happens-before order [8] に反しない順序で永続化する．happens-before order は，Java のメモリモデルで定義された，複数のスレッドによる書き込みがメモリに反映されるときに満たすべき半順序である．AutoPersist と異なり，RBP は永続化対象となった DRAM 上のオブジェクトの**複製**を NVM 上に作成することで永続化を実現する．複製を持つオブジェクト（これを永続オブジェクトと呼ぶ）の DRAM 上のオブジェクトは，無効化せずに使い続ける．永続オブジェクトへの書き込みは DRAM と NVM の両方に行い，複製は常に最新の値を保持する．全てのオブジェクトは永続化の対象であるかどうかに関わらず DRAM 上にオブジェクトが存在するため，読み出しは常に DRAM から行うことが可能である．

RBP で動作させる Java プログラムは，データレースフリーであることを仮定している．つまり，複数のスレッドが同じオブジェクトのフィールドに競合するアクセスを行わないよう，Java プログラムは適切に排他制御を行う必要がある．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例外である．通常，`volatile` フィールドは他のスレッドとの同期に使用されるため，RBP では複数のスレッドが `volatile` フィールドに同時にアクセスできるよう設計されている．

上記のようにプログラムがデータレースフリーであることを仮定しても，以下のような場合に競合が発生する．

- オブジェクトの複製を作成中に，別のスレッドがそのオブジェクトに書き込む．
- 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する．
- 複数のスレッドが同じオブジェクトの複製を作成する．

これらを考慮してアルゴリズムを設計しなければ，複製が古いデータで作成されたり，有効な複製が複数個作成されたりする可能性がある．その結果，システムのクラッシュ後に NVM 上に残ったオブジェクトのデータが不整合な状態となる．

本研究では，これらの競合を解決して，以下の二つを保証するアルゴリズムを提案する．

- 永続ルートから到達可能な全てのオブジェクトは永続化されている．

- 全ての永続オブジェクトは元のオブジェクトが DRAM 上に，複製が NVM 上にあり，書き込みが他のスレッドに観測されるか，同じスレッドによる次の書き込みが行われる前に，これらの内容は同期されている．

元のオブジェクトと複製の内容が同期しているというのは，オブジェクトの各フィールドが以下の条件を満たすことを意味する．

- プリミティブ型の場合，DRAM 上のオブジェクトと NVM 上のオブジェクトの両方に同じ値が書き込まれている．
- 参照型の場合，DRAM 上のオブジェクトには参照先（DRAM 上にあるオブジェクト）を，NVM 上のオブジェクトには参照先の複製（NVM 上にあるオブジェクト）を指す参照が書き込まれている．

RBP は通常のフィールドへの書き込みでロックを使わず，細粒度の同期処理を用いる．`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` フィールドへの読み書き操作は例外であり，これらにはスピンロックを用いる．

RBP が複製を作成するための領域は，NVM 専用のメモリアロケータにより確保する．不要な NVM 上のオブジェクトを回収するごみ集め（GC，ガベージコレクション）の機能には，本研究と並行して長安らにより開発された RBP 向けの GC [9] を用いる．

以上のような永続化機構によって，永続オブジェクトのデータはクラッシュが発生しても消えず NVM に残るが，DRAM 上にある永続ルート，DRAM 上のオブジェクトのデータ，NVM 領域管理のためのメタデータは消えてしまう．この状態では Java プログラムから永続オブジェクトにアクセスすることができず，永続データを利用したプログラムの再開はできない．本研究では，この問題を解決するリカバリ機構を Java API として提供する．リカバリ機構は，クラッシュ前にリカバリに必要なデータを NVM 上に永続化しておき，クラッシュ後にその NVM 上のデータを元に DRAM 上のデータを再構築してクラッシュ前の状態に戻す．Java プログラマは，リカバリ機構の Java API をプログラム再起動時に呼び出すようにプログラムを記述する．

リカバリ機構を実現するために，リカバリに必要なクラスのクラス名などのメタ情報と永続ルートも永続化する．リカバリは，初めに全ての永続オブジェクトのための DRAM 領域を確保し，その後に各永続オブジェクトの内容を NVM 領域から DRAM 領域にコピーすることで実現する．ただし，参照値はコピー前に DRAM 中のアドレスへの変換が必要となる．また，リカバリ中のクラッシュや GC に対処しなければならない．リカバリ処理の一部は Java ライブラリメソッドとして実装し，C++で実装したランタイムシステムの関数と協調して動作することで，Java クラスや GC に関する煩雑な処理を簡略化する工夫を行った．

評価のために，これらを実用的な Java 仮想機械（VM）である OpenJDK 16 [2, 3] に実装した．AutoPersist の書き込み処理の一部も実装して，オブジェクトを全く永続化しないアプリケーションプログラムにかかるオーバヘッドと，オブジェクトを永続化する際のオーバヘッド（オブジェクトを永続化して複製を同期し続けるコスト）を測定した．実験結果より，複製を作成するオーバヘッドは小さく，永続オブジェクトから読み出すオーバヘッドはほぼ存在しないことがわかった．永続オブジェクトへの書き込みのオーバヘッドは，ほとんどが NVM への書き込みによるものであった．オブジェクトを永続化しないプログラムのオーバヘッドは，先行研究である AutoPersist と同程度であることがわかった．また，正しくリカバリができるか確認するために，複数の Java プログラムを用いて実行中に計算機をクラッシュさせる動作確認を行った．さらに，RBP を用いる Java アプリケーションの作成手順を示し，リカバリに必要な時間を調査した．

第 2 章

前提知識

2.1 不揮発性メモリ

不揮発性メモリ（Non-volatile memory，NVM）は，電源を喪失してもデータを保持し続けることができる主記憶装置の総称である．従来の揮発性メモリである DRAM と同様に，ロード・ストア命令を用いて容易にバイトアクセスが可能である．従来の永続化デバイスである HDD や SSD よりアクセスが高速に行えるが，その性能は DRAM より劣る．容量単価が安価であり，データの保持に電源を必要としないことから消費電力が少ない点でも注目されている．

Intel Optane DC Persistent Memory は，NVM の代表的な製品である [1]．この製品には，大容量の揮発性メモリとして利用する Memory Mode と，不揮発性メモリとして利用する AppDirect Mode がある [10]．Memory Mode は，OS に対して通常の主記憶装置として振る舞う．書き込んだデータは揮発性であり，クラッシュ後にデータが残らない．NVM の管理は OS の制御下にあり，アプリケーション側での対応は必要ない．また，メモリコントローラは DRAM を NVM のキャッシュとして利用する．AppDirect Mode は，OS に対してデバイスとして振る舞う．書き込んだデータは不揮発性であり，クラッシュ後もデータが残る．ただし，NVM を制御するシステムの導入や，アプリケーション側での対応が必要である．

本研究では，AppDirect Mode で NVM をデバイスとして認識させる．さらに，DAX（Direct Access）をサポートする NVM 用のファイルシステムである NOVA [11, 12, 13] を組み合わせて利用する．NOVA は，ブロックサイズの粒度で書き込みを行うことが前提で

ある既存のファイルシステムと異なり，キャッシュラインサイズの粒度で書き込みを行うことが可能な NVM に最適化している．また，Linux カーネルのページキャッシュ機能等の中間処理を削除することで，アクセス時のオーバヘッドを最小限にする．本研究では，NVM 領域をマウントして事前に十分なサイズのファイルを作成しておき，RBP はその抽象化された NVM ファイルを `mmap` システムコールで仮想メモリ空間にマップして利用する．

2.2　キャッシュラインの書き戻しとメモリバリア

NVM への書き込みは DRAM と同様に揮発性の CPU キャッシュを介して行われるため，書き込み命令を実行しても実際には CPU キャッシュにだけ書き込まれていて NVM には書き込まれていないような過渡的な状態がある．つまり，NVM への書き込みは，書き込んだデータの永続性を即座に保証するものではない．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．永続化されたデータの整合性を維持するためには，キャッシュラインを明示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．

Intel x64 アーキテクチャの CPU [14] は，`CLWB`（`Cache Line Write Back`）命令を持っている．この命令は，CPU 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` 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ため，`CLWB` 命令だけでは意図しない順序でキャッシュラインが書き戻される可能性がある．キャッシュラインが実際に NVM に書き戻されることを保証するには，後続の順序付けしたい書き込みの前にメモリバリアである `SFENCE` 命令や `MFENCE` 命令などを挿入し，`CLWB` 命令の完了まで後続の書き込み命令を遅らせる必要がある．

メモリバリアは，メモリ操作を直列化する．`SFENCE` 命令の前にある書き込み命令は，後続の書き込み命令より前に完了する．`MFENCE` 命令の前にある書き込み・読み込み命令は，後続の書き込み・読み込み命令より前に完了する．`CLWB` 命令は `SFENCE` 命令で順序付けが可能であるため，本研究では `CLWB` 命令と `SFENCE` 命令を用いたキャッシュラインの書き戻

しを行う．また，書き込み命令と読み出し命令を順序付けするために MFENCE 命令を用いる．

2.3 OpenJDK 16

OpenJDK は，Java プログラムの実行と開発を行う環境を提供する JDK（Java Development Kit）のオープンソース実装である．OpenJDK の VM に関する処理は主に C++ で記述されており，ソースコードを書き換えてビルドすることで処理の追加や変更を行うことが可能である．以降の節では，VM の処理を書き換えるために必要な OpenJDK の前提知識について述べる．

2.3.1 Template Interpreter

Java プログラムは中間言語であるバイトコードにコンパイルされ，それを Java VM が読み込んで実行する．OpenJDK 16 では，デフォルトで Template Interpreter と呼ばれるインタプリタによって，バイトコードの実行が行われる．Template Interpreter は，Java VM の起動時に生成されるインタプリタである．実行するマシンの OS や CPU アーキテクチャに応じて各バイトコードを実行する機械語命令列を生成し，それをインタプリタとして用いる．

Template Interpreter にはスレッデッドコード [15] と呼ばれる技法が使われている．各バイトコードを実行する機械語命令列のエントリーポイントは，dispatch table と呼ばれる配列に格納されている．各バイトコード実行の最後で，次に実行するバイトコードを読み取って対応するエントリーポイントにジャンプする処理を行うことで，順にバイトコードを実行する．

2.3.2 バリアセット

本節の内容は，2.2 節で述べたメモリバリアとは無関係であることに注意されたい．

バリアセットは，Java ヒープに対する読み書き操作の処理をまとめたものである．主に

GC 処理の補助をする目的で利用されており，各 GC に対応するバリアセットが用意されている．バリアセットには，ランタイム用，インタプリタ用，C1 JIT コンパイラ用，C2 JIT コンパイラ用の 4 種類が実装されている．バリアセットのうち，Java ヒープに対する書き込みを行う際に決められた処理を呼び出す仕組みをライトバリア，Java ヒープに対する読み出しを行う際に決められた処理を呼び出す仕組みをリードバリアと呼ぶ．

バリアセットのインタプリタ用（Template Interpreter）処理には，Java ヒープに書き込みを行うバイトコードである `put_field`，`put_static`，`aastore` などを実行する機械語命令列に，ライトバリアが織り込まれている．

Template Interpreter によるバイトコードの実行だけではなく，VM の内部処理においても Java ヒープへのアクセスが行われる．VM の内部処理から Java ヒープへアクセスする場合は，ランタイム用のバリアセット（C++で記述された専用の関数）を介して行われる．ただし，GC などの一部の処理は，バリアセットを利用せずにアクセスすることがある．

多くの GC では，このバリアセットの仕組みを用いて，参照型の値をオブジェクトに書き込む際に追加の必要な処理を実行する．しかし，OpenJDK 16 には整数値などの参照型以外の値のアクセスにバリアセットを適用する仕組みも用意されている．

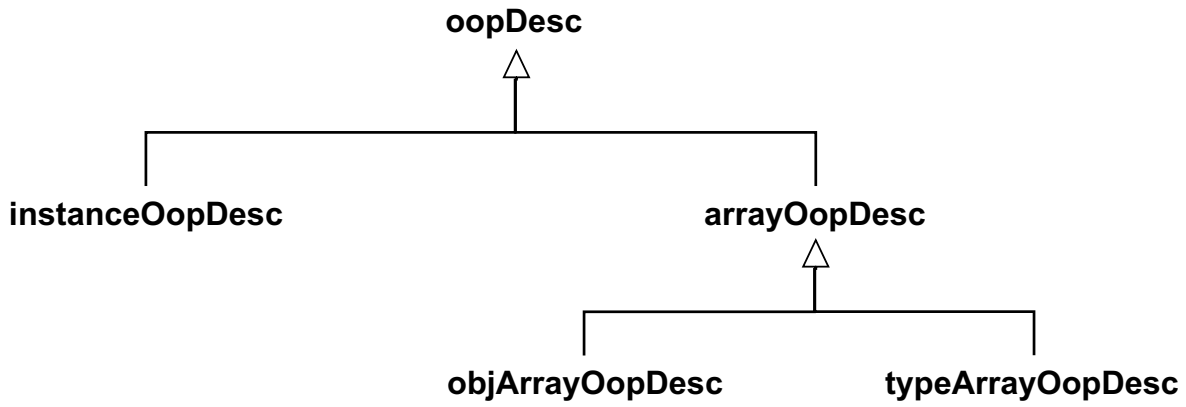


図 2.1　オブジェクト構造体を定義するクラスの継承関係

2.3.3 オブジェクト構造体

OpenJDK 16 における GC 対象のオブジェクトの構造体は，oopDesc クラスを基底クラスとする形で定義されている．図 2.1 は，オブジェクト構造体を定義するクラスの継承関係を示したものである．instanceOopDesc クラスは Java の配列以外のオブジェクト，arrayOopDesc クラスは Java の配列オブジェクトについて定義している．Java の配列オブジェクトに関する定義は要素の型によって更にクラスが分けられており，要素が参照型である場合は objArrayOopDesc クラス，要素がプリミティブ型である場合は typeArrayOopDesc クラスを用いる．

図 2.2 は，instanceOopDesc クラスと arrayOopDesc クラスで定義されるオブジェクト構造体を示したものである．基底クラスである oopDesc クラスによって，先頭の二つのフィールドが定義されている．markWord は oopDesc クラスを継承する形で定義されているクラスであるが，実体は先頭 1 ワードへのアクセサメソッドを持つクラスである．このワードの各ビットを用いて，GC やロックに関するデータを保持している．次に配置されている Klass クラスの参照は，このオブジェクトがどの Java のクラス（以下，Java クラスと呼ぶ）に属すのかを示す．Klass クラスは Java クラスに関する情報を保持する C++の

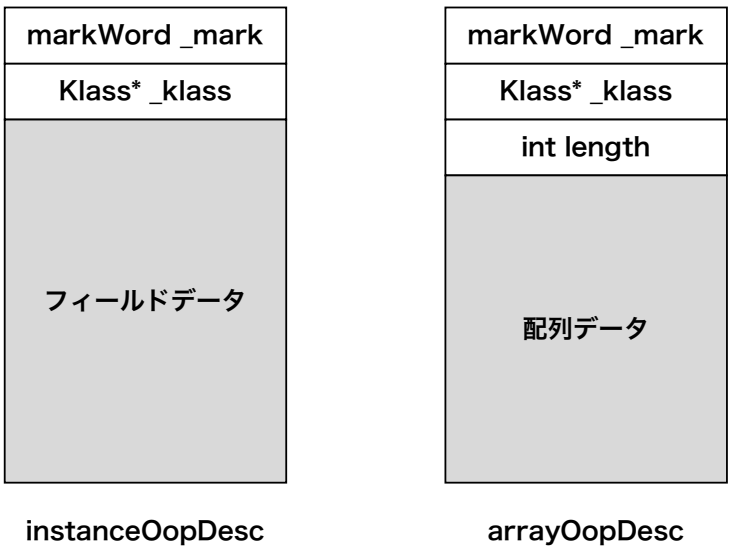


図 2.2 オブジェクト構造体

クラスであり，各 Java クラスごとに GC 対象外のメモリ領域に作成される．

Java の配列以外のオブジェクトを表す `instanceOopDesc` クラスは，`oopDesc` クラスのオブジェクトヘッダに続いてフィールドのデータが配置される．フィールドデータの領域は，そのオブジェクトが属するクラスによってサイズや配置位置が異なる．Java の配列を表す `arrayOopDesc` クラスは，`oopDesc` クラスのオブジェクトヘッダに加えて，配列の要素数を示す int 型の値 `length` が配置される．それに続いて，配列の各要素のデータが配置される．Java プログラムで `new` キーワードを使用すると，先述のオブジェクト構造体の定義に従ってオブジェクトが生成される．

2.3.4 Java クラス情報

図 5.1 に，OpenJDK と既存の永続化機構が作成するデータを表す．斜線柄の四角形は Java 言語上のクラスを実装するオブジェクトであるクラスオブジェクト [*1] を，ドット柄の四角形はランタイム内部で Java クラス情報を保持するデータ Klass を示す．なお，DRAM 上のオブジェクトを dram copy，NVM 上のオブジェクトを nvm copy と表記する．また，クラスオブジェクトとそれ以外を区別するため，オブジェクトの中でも DRAM 上のクラスオブジェクトを dram class，NVM 上のクラスオブジェクトを nvm class と表記する．

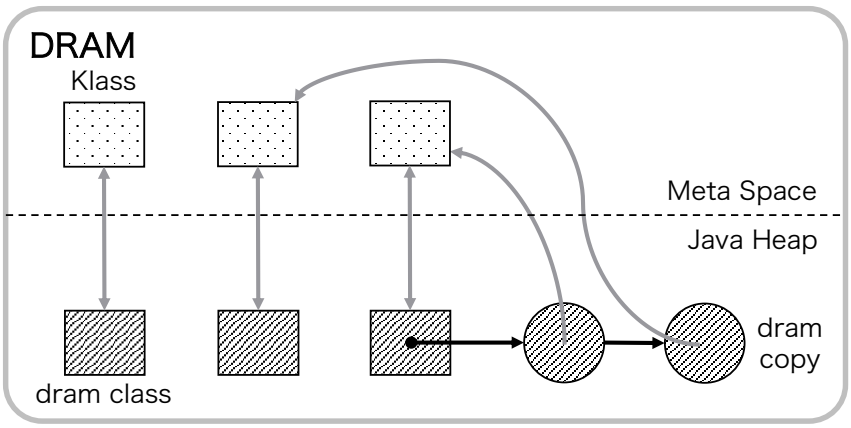


図 2.3 OpenJDK が作成するデータ

[*1] OpenJDK では Mirror オブジェクトと呼ばれているが，名前が紛らわしいのでクラスオブジェクトと呼ぶ．

```
1  Class<?> mirror1 = Main.class;
2  Class<?> mirror2 = new Main().getClass();
```

図 2.4 Main クラスのクラスオブジェクトを取得する Java プログラム

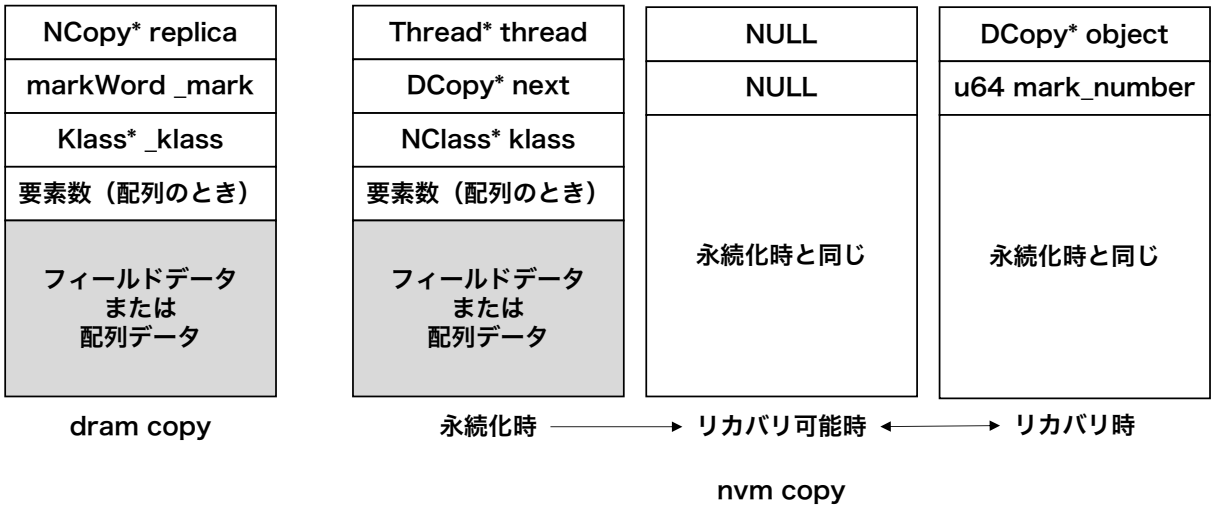


図 2.5 dram copy と nvm copy のデータ構造

本論文では，クラスオブジェクトは Java プログラムにおける Java クラス表現である `java.lang.Class` クラスのインスタンスを指す．これは，ランタイムによって全ての Java クラスに対して一つずつ自動的に作成され，図 2.4 のようにクラスリテラル（`.class`）または `getClass` メソッドを用いて取得できる．OpenJDK 16 では，static フィールドのデータがクラスオブジェクトの末尾に格納されている．これらは，Java のプログラムからクラスオブジェクトのフィールドとしてアクセスすることはできない．クラスオブジェクトは，他のオブジェクトと同じく GC の対象である Java Heap 領域に配置される．

Klass は，OpenJDK におけるランタイム内部の Java クラス表現である．全ての Java クラスに一つずつ作成され，Klass とクラスオブジェクトは互いに参照し合うようになっている．dram copy は，そのオブジェクトが属す Java クラスの Klass への参照をヘッダに保持している．Klass は，GC の対象外である DRAM 上の Meta Space 領域に配置される．

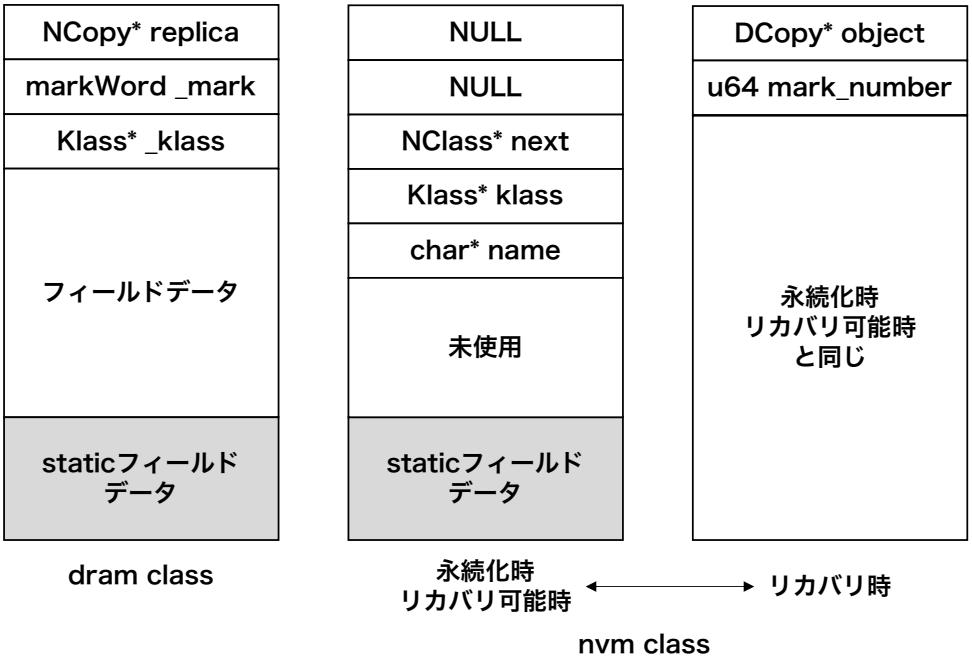


図 2.6 dram class と nvm class のデータ構造

2.3.5 オブジェクトの拡張と複製のデータ構造

本節では，RBP を実装する上で必要なヘッダを拡張したオブジェクトと，その複製のデータ構造について述べる．ただし，各データの詳細は 4 章および 5 章にて後述する．

図 2.5 は，dram copy と nvm copy のデータ構造の概略である．本研究では，オブジェクトヘッダの先頭に 1 ワードの領域を追加する．dram copy の先頭ワードである `NCopy*` 型の `replica` フィールドは，nvm copy への参照である．先頭ワード以外の領域は，2.3.3 節で述べた元々の用途で利用する．一方，nvm copy にも同様に 1 ワードの領域を追加するが，先頭の 3 ワードは dram copy と異なる用途で利用する．また，同じ領域を他の処理と共有で利用するため，状態によって用途が異なる．

永続化時（複製の作成時）には，次のような用途で利用する．`Thread*` 型の `thread` フィールドは，4.4.2 節と 4.4.6 節で述べる担当スレッドを指す参照である．`DCopy*` 型の `next` フィールドは，4.4.6 節で述べる同じ担当スレッドによって作成された nvm copy のリストを構成するための参照である．`NClass*` 型の `klass` フィールドは，5.1.2 節で述べる

nvm class を指す参照である．

リカバリ時には，次のような用途で利用する．`DCopy*` 型の `object` フィールドは，5.3.4 節で述べる dram copy を指す参照である．`u64` 型の `mark_number` フィールドは，5.4 節で述べる符号なし 64 ビット整数のマーク番号である．

図 2.6 は，dram class と nvm class のデータ構造の概略である．dram class のオブジェクトヘッダの用途は dram copy と同様であるため，説明を省略する．nvm class は，一部のデータ構造が nvm copy と異なる．`Klass*` 型の `klass` フィールドは，5.3.1 節で述べる Klass を指す参照である．`char*` 型の `name` フィールドは，5.1.2 節で述べるクラス名を指す参照である．また，フィールドデータは永続化の対象外であり，それに対応する領域は nvm class では未使用領域となる．

第 3 章

プログラミングモデル

3.1　到達可能性に基づく永続化

Orthogonal persistency [4] を構成する原則の一つに，“永続化するオブジェクトの指定はシステムに直交している” というものがある．この原則に対して，我々は到達可能性に基づく永続化モデルを用いる．これは，プログラマが指定した永続ルート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，永続化の対象となる．

図 3.1 は，到達可能性に基づく永続化モデルにおいて，永続化の対象であるオブジェクトを斜線柄の円，そうでないオブジェクトを白色の円で表している．この永続化モデルは，プログラマが永続化対象のオブジェクトを一つずつ指定する必要はなく，永続ルートを指定するだけで良いというメリットを持つ．全てのオブジェクトは，作られたときは永続化の対象外である．オブジェクトへの参照が永続ルートまたは永続オブジェクトの中のフィールドに格納されると，そのオブジェクトは永続化の対象になる．

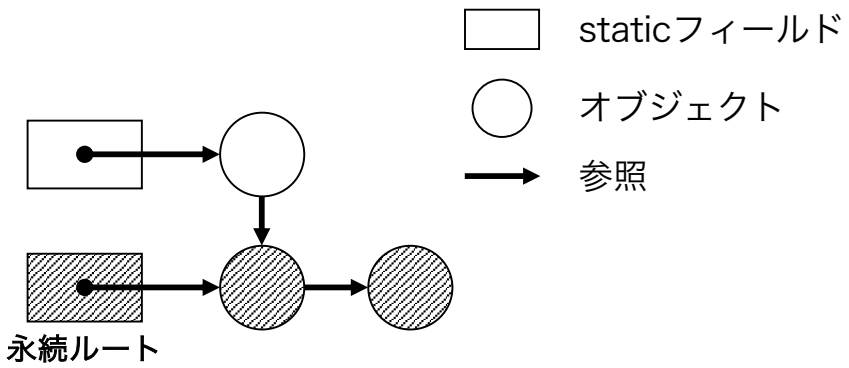


図 3.1　到達可能性に基づく永続化モデル

```
1  import rbp.*;
2
3  @durableroot public static Object root;
4
5  public static void main(String[] args) {
6    Recovery.setNvmFile("/path/to/nvm");
7
8    if (Recovery.hasValidData()) {
9      ClassLoader[] clds = { ... };
10     Recovery.recovery(clds);
11   } else {
12     Recovery.init();
13   }
14 }
```

図 3.2 API の使用例

3.2 RBP が提供する Java API

本節では，RBP が提供する Java API について記す．この API は，オブジェクトのデータをリカバリ可能にするシステムの実装において必要となる．

API の使用例を図 3.2 に示す．1 行目でインポートするパッケージには，RBP が提供する API が含まれている．

RBP は，永続化機構の API として `durableroot` アノテーションを提供する．これは，Java プログラマが 3.1 節で述べた永続ルートを指定するためのものであり，参照型の static フィールドを定義する部分に記述する．3 行目は，`durableroot` アノテーションを付けて Java プログラマが指定した永続ルートである．プログラム起動直後の値は `null` である．

また，リカバリ機構の API として，`Recovery` クラスを提供する．`Recovery` クラスは，主に以下の 4 つのメソッドを提供する．

- `void setNvmFile(String filepath);`
- `boolean hasValidData();`
- `void recovery(ClassLoader[] clds);`

- `void init();`

Java プログラマは任意のタイミングでこれらを呼び出すことができるが，典型的にはプログラムの起動直後に呼び出す．

6 行目の `setNvmFile` メソッドは，RBP が使用する NVM ファイルのパスを指定する．このメソッドは，他のメソッドを呼び出す前に必ず実行しなければならない．

`recovery` メソッドと `init` メソッドは，どちらか一方のみを実行する．`recovery` メソッドは有効な永続データが存在する場合にのみ実行可能であり，リカバリ処理を行う．`init` メソッドは有効な永続データの有無に関わらず実行可能であり，NVM 上のデータを初期化する．

8 行目の `hasValidData` メソッドは，有効な永続データの有無を返す．これは，`recovery` メソッドと `init` メソッドのどちらの処理を行うか判断するために利用する．

9 行目と 10 行目は，リカバリ処理を行う場合に実行される．`recovery` メソッドには，使用する全てのクラスローダーを引数に与える．`recovery` メソッドを実行すると，元々 `null` であった 3 行目の永続ルートに，リカバリされた永続オブジェクトへの参照が格納される．

12 行目は，リカバリを行わずに NVM 上のデータを初期化する場合に実行される．`init` メソッドを実行すると，NVM 上にある既存のデータを破棄した後に，4.5 節で後述する NVM アロケータの初期化と，5.1.1 節や 5.1.2 節で後述するクラスオブジェクトの複製（nvm class の作成）を行う．

`recovery` メソッドまたは `init` メソッドを実行した後に，永続化機構が動作を開始する．

第 4 章

永続化機構

4.1 基本的なアイデア

3.1 節で示した到達可能性に基づく永続化を，リードバリアを使用せずに実現する RBP アルゴリズムを提案する．

AutoPersist のような移動アプローチは，非永続オブジェクトが永続化対象になった直後に，DRAM 上のオブジェクト（dram copy と呼ぶ）を NVM 上に**移動**させる．移動先の NVM 上にあるオブジェクト（nvm copy と呼ぶ）は，nvm copy 領域を割り当てた後に dram copy の内容を nvm copy にコピーすることで作られる．dram copy には nvm copy への転送ポインタをセットし，dram copy を無効化する．一つのオブジェクトには dram copy と nvm copy のどちらか一方のみが対応しており，非永続オブジェクトは dram copy で表され，永続オブジェクトは nvm copy で表される．

例えば，図 4.1 のオブジェクト A，B，C を含むヒープを考える．x と y は変数であり，斜線は永続オブジェクトであることを示している．y は永続ルートであり，B と C は永続オブジェクトである．

図 4.2 (a) は，B への参照が y に格納された直後の AutoPersist のヒープの様子である．点線の円で表される B と C は移動前の古い領域であり，点線の矢印で示す転送ポインタを持つ．古い領域への参照を用いた B と C へのアクセスは，転送ポインタを介して新しい領域に転送される．ヒープ内のオブジェクトには古い領域への参照と新しい領域への参照が混在しているため，`if_acmpeq` などの比較命令も転送ポインタを考慮する必要がある．

一方，RBP は非永続オブジェクトの複製を NVM 上に作成して永続化する．つまり，

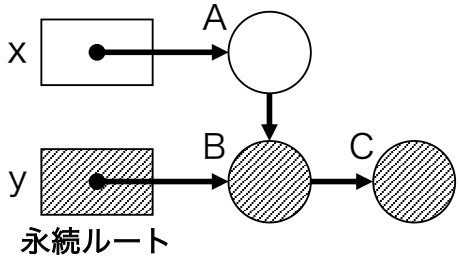
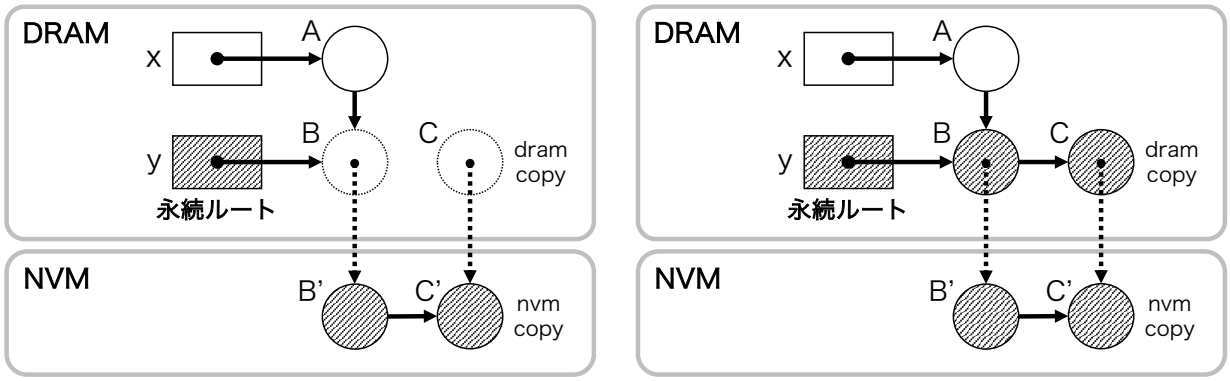


図 4.1　Java プログラムから見たヒープの例



(a) AutoPersist　　(b) RBP

図 4.2　永続化手法

DRAM 上にあるオブジェクト（dram copy）の複製（nvm copy）を NVM 上の領域に作成することで永続化を実現する．これは，RBP が DRAM 上のオブジェクトである dram copy を無効化しないことを意味する．二つの領域を対応付けるために，1 ワードの領域をオブジェクトのヘッダに追加して，dram copy のヘッダに nvm copy への転送ポインタを持たせる．この転送ポインタを書き込む dram copy のフィールドを，`replica` フィールドと呼ぶ．永続オブジェクトの dram copy と nvm copy は，プリミティブ型のフィールドに同じ値を持ち，参照型のフィールドには同じオブジェクトの dram copy と nvm copy を指す参照をそれぞれ持つことで緩やかに同期される．したがって，永続ルートから到達可能な全ての nvm copy は，dangling pointer を作ることなくクラッシュ後に復元可能である．

図 4.2 (b) は，B への参照が y に格納された直後の RBP のヒープの様子である．全てのオブジェクトは，DRAM 上に dram copy を持っており，永続オブジェクトは NVM 上に nvm copy も持っている．図中の B と C は dram copy であり，B' と C' は nvm copy であ

る．B と C は各々 B' と C' に対応しており，点線の矢印で示す転送ポインタを持つ．Java プログラム内の参照は，常に dram copy への参照によって表される．つまり，全ての参照型（プリミティブ型以外）の変数には，dram copy への参照または `null` が格納される．

読み出しは永続オブジェクトであるか否かに関わらず，リードバリアを使わずに dram copy から値を取得する．一方，dram copy と nvm copy の両方に同じ最新の値を持たせて整合性を維持するため，永続オブジェクトへの書き込みは dram copy と nvm copy の両方に対して行う．すなわち，dram copy はアクセスの遅い nvm copy のキャッシュのような働きをする．この二箇所への書き込みを**ダブルアップデートバリア**と呼び，ライトバリアを改造することで実現する．二箇所への書き込みをアトミックに行うことはできず，各々の処理が完了するまでに時間差が生じるが，データレースフリーを仮定しているため，nvm copy への書き込みが完了する前に dram copy の値が読み出されることはない．

永続オブジェクトに書き込む参照の参照先が非永続オブジェクトである場合は，先にそのオブジェクトの複製を NVM 上に作成して永続化する．これにより，永続オブジェクトのフィールドが保持する参照は常に永続オブジェクトのみを指すことが保証される．複製の作成は，4.5 節で述べる NVM アロケータにより nvm copy 領域を確保した後に，dram copy から nvm copy にフィールドの内容をコピーすることで行う．このコピー処理で nvm copy に非永続オブジェクトへの参照を書き込む場合は，再帰的に参照先のオブジェクトの複製を作成して永続化した後に，nvm copy への参照を書き込む．

4.2　オブジェクト色と不変条件

オブジェクトを永続化する過程を説明するために，オブジェクトの永続化状態を示す概念を導入する．各オブジェクトの状態は **白**，**灰**，**黒**，**青** の四色のうち，いずれかの色で表す．このオブジェクトの状態は説明のために導入するものであり，実際のオブジェクトには状態を保持するためのフィールドが存在しないことに注意されたい．

白オブジェクトは nvm copy を持たない．

表 4.1　オブジェクトの状態と `replica` フィールドの関係

状態	`replica` フィールド		
	転送ポインタの書き込み	整合性のある nvm copy への参照	nvm copy のキャッシュライン書き戻し
白	no	–	–
灰	yes	no	–
黒	yes	yes	no
青	yes	yes	yes

灰オブジェクトは，**同期されていない**（dram copy とは内容が異なる）nvm copy を持つ可能性がある．**灰オブジェクト**が指すオブジェクトには，nvm copy がまだ存在していない可能性がある．

黒オブジェクトには，dram copy と同期している nvm copy があり，その nvm copy が配置されたメモリ番地に対して `CLWB` 命令を発行しているが，まだ完了していない可能性がある．**黒オブジェクト**が指す全てのオブジェクトは，nvm copy を持っている．

青オブジェクトは，キャッシュラインが書き戻されて，データが NVM に永続化された nvm copy を持つ．また，**青オブジェクト**が指す全てのオブジェクトは，キャッシュラインが書き戻された nvm copy を持つ．

最初の三色は，GC の三色抽象化 [16] に類似している．表 4.1 は，オブジェクトの状態と，そのオブジェクトの dram copy の `replica` フィールドとの関係をまとめたものである．状態が**青**のとき，オブジェクトはクラッシュから復元可能である．

オブジェクトの複製を NVM 上に作成する過程を，このオブジェクト色の抽象化を用いて説明する．まず，nvm copy 領域を確保して，dram copy に nvm copy 領域への転送ポインタを書き込む．このとき，**白オブジェクト**から**灰オブジェクト**に変化する．**灰オブジェクト**は nvm copy を持つが，その nvm copy のフィールドにはデータが書き込まれていない．次に，フィールドのデータを順に書き込み，`CLWB` 命令を発行してキャッシュラインの書き

戻しを開始する．このとき，**灰オブジェクト**から**黒オブジェクト**に変化する．キャッシュラインの書き戻しが完了すると，オブジェクトは**黒オブジェクト**から**青オブジェクト**に変化して，復元可能になる．

提案アルゴリズムは，最初の三色のオブジェクトについて**強い三色不変条件**（*the strong tri-color invariant*）を保証する．

不変条件 1 黒オブジェクトは**白オブジェクト**を参照しない．

さらに，このアルゴリズムは**青オブジェクト**に関する別の不変条件も保証する．

不変条件 2 青オブジェクトの *nvm copy* は，**青オブジェクト**の *nvm copy* のみを参照する．

さらに，ライトバリアによって書き込まれている途中のオブジェクトを除き，次の不変条件が保証される．

不変条件 3 黒オブジェクトまたは **青オブジェクト**は，内容が一致している *dram copy* と *nvm copy* を持つ．

不変条件 4 青オブジェクトの *nvm copy* の内容は，*NVM* に書き込みが完了している．

不変条件 3 と 4 より，**青オブジェクト**は dram copy と一致する内容を持つ nvm copy がクラッシュ後にも NVM に確実に残ることが保証される．クラッシュが発生した瞬間にライトバリアで書き込みが行われていたオブジェクトは，dram copy のみに新しい値が書き込まれた状態である可能性がある．ただし，データレースフリーを仮定しているため，他のスレッドからこの値を読み出すことはできない．したがって，dram copy への書き込みを行う前にクラッシュが発生したと見做すことが可能である．不変条件 2 より，リカバリ後に dangling pointer が作られることはない．

4.3　シングルスレッド実行

本節では，シングルスレッド環境を仮定し，複数のスレッド間の競合を解決する必要がない単純化したバージョンの RBP について説明する．後ほど，マルチスレッド環境でも動作するように 4.4 節で修正する．

4.3.1　永続化処理

図 4.3 は，永続化処理のエントリポイントである `make_persistent` の擬似コードである．ここでは，dram copy への参照を `DCopy` 型，nvm copy への参照を `NCopy` 型，オブジェクト `o` のフィールド `f` を `o[f]` と表記する．また，簡単化のためにオブジェクトのフィールドは全て参照型であると仮定する．シングルスレッド環境であるため，`make_persistent` の処理が完了した後のオブジェクトは全て**白オブジェクト**または**青オブジェクト**のいずれかの状態であり，**灰オブジェクト**や**黒オブジェクト**のような過渡的な状態は存在しない．

永続化するオブジェクトの dram copy を `r` とする．まず，`make_persistent` は `shade` を用いて `r` を**灰オブジェクト**にする（4 行目）．**灰オブジェクト**にするというのは，NVM 上に nvm copy の領域を割り当て，dram copy の `replica` フィールドに複製である nvm copy への転送ポインタを書き込むことを意味する．**灰オブジェクト**の集合はワークリストを用いて管理している．5 行目では，ワークリストに**灰オブジェクト**を挿入している．

次に，ワークリストの**灰オブジェクト**が無くなるまで，**灰オブジェクト**を**黒オブジェクト**にする処理を繰り返す．具体的には，ワークリストから**灰オブジェクト**を取り出し（7 行目），オブジェクト `o` の各フィールド `f` に対して，以下のような処理を行う．

1. フィールド `f` から参照している**白オブジェクト** `p` を，`shade` によって**灰オブジェクト**にする（12 行目）．
2. オブジェクト `p` の nvm copy への参照を，オブジェクト `o` の nvm copy のフィールド `f` に書き込む（13 行目）．

```
1  // 永続化処理のエントリポイント
2  // オブジェクト r を永続化する（つまり，青色にする）
3  make_persistent_single(DCopy r) {
4    if (shade(r) == TRUE) {   // オブジェクト r を白色から灰色にする
5      worklist.add(r);
6      while (not worklist.empty()) {
7        DCopy o = worklist.remove();
8        NCopy on = o.replica;
9        // オブジェクト o は灰色
10       foreach (Field f in o.fields) {
11         DCopy p = o[f];
12         if (shade(p) == TRUE) worklist.add(p);
13         on[f] = p.replica;
14       }
15       CLWB_RANGE(on, on.size());
16       // オブジェクト o は黒色
17     }
18     SFENCE;
19   }
20 }
21
22 // オブジェクト o が白色ならば灰色にして TRUE を返す
23 shade_single(DCopy o) {
24   if (o.replica != NULL) return FALSE;  // オブジェクト o は白色ではない
25   o.replica = alloc_NVM(o.size());      // nvm copy の領域を確保する
26   return TRUE;
27 }
```

図 4.3　シングルスレッド版 make_persistent とシングルスレッド版 shade

全てのフィールドを処理した後に，CLWB 命令（15 行目）を発行して，オブジェクト o の nvm copy の領域全体のキャッシュラインを書き戻す．ここでは，CLWB 命令による書き戻しが完了するまで待機しないことに注意されたい．このとき，オブジェクト o は**黒オブジェクト**になる．全てのオブジェクトが**黒オブジェクト**になると，SFENCE 命令を発行して（18 行目），全ての**黒オブジェクト**が同時に**青オブジェクト**に変化する．

```
1  // オブジェクト o の参照型フィールド f に，オブジェクト v への参照を書き込む
2  putfield_single(DCopy o, Field f, DCopy v) {
3    NCopy on = o.replica;
4    if (on == NULL) {
5      // オブジェクト o は 白色
6      o[f] = v;
7      return;
8    }
9    // オブジェクト o は 青色
10   make_persistent(v);
11   o[f] = v;
12   on[f] = v.replica;
13   CLWB(&on[f]);  // on[f] のキャッシュラインを書き戻す
14   SFENCE;
15 }
```

図 4.4　シングルスレッド版 putfield

4.3.2　ライトバリア

図 4.4 は，オブジェクト o の参照型フィールド f にオブジェクト v を書き込む putfield の擬似コードである．4.2 節で示した不変条件を維持するために，二種類のライトバリアを利用する．プリミティブ型の値は書き込み時にオブジェクトを永続化する必要がないため putfield は単純であり，ここでは説明を省略している．

まず，putfield は**挿入バリア**（*insertion barrier*）[16] を用いて不変条件 2 を維持する．オブジェクト o の replica フィールドが NULL ではない場合，オブジェクト o は**青オブジェクト**である．**青オブジェクト** o のフィールド f に書き込みを行う前に，putfield は make_persistent を呼び出してオブジェクト v を**青オブジェクト**にする（10 行目）．

次に，putfield は**ダブルアップデートバリア**を用いて，不変条件 3 を維持する．このバリアは，オブジェクト v への参照をオブジェクト o のフィールド f に書き込み（11 行目），オブジェクト v の nvm copy への参照をオブジェクト o の nvm copy の同じフィールドに書き込む（12 行目）．さらに，不変条件 4 を維持するために，CLWB 命令と SFENCE 命令（13–14 行目）を発行して，AutoPersist [17, 6] と同様に nvm copy のフィールド f の

キャッシュラインを書き戻す.

4.4　マルチスレッド実行

4.3 節で示したアルゴリズムでは，マルチスレッド環境の実行において競合状態となる可能性がある．本節では，マルチスレッド環境での実行を考慮したアルゴリズムを説明する．

RBP で動作させる Java プログラムは，データレースフリーを仮定している．そのため，永続オブジェクトへの読み書きにおいて，競合状態は発生しない．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例外である．これらの例外については，4.4.7 節で説明する．

4.4.1　競合状態

マルチスレッド環境では，以下のような競合状態が発生する可能性がある．

Race 1: 複数のスレッドが同時に `make_persistent` を呼び出し，同じオブジェクトの複製を作成する．

Race 2: あるスレッドがオブジェクトに対して `putfield` を実行している間に，別のスレッドがそのオブジェクトの複製を作成する．

Race 3: あるスレッドが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 `putfield` を実行する．

Java プログラムがデータレースフリーであるという仮定より，複数のスレッドが同時にオブジェクトの同じフィールドに対して `putfield` を実行することによる競合は発生しない点に注意されたい．

例えば，以下のようなプログラムを実行することを考える．`u` は永続ルートの一つである．

```
u = x;
```

u は永続ルートであるため，このスレッドはオブジェクト x の複製を作成して永続化する．一方，別のスレッドは以下に示すようなオブジェクト x に対して putfield を実行するプログラムを動作させることが可能である．

```
x.f = 1;
```

前者のスレッドがオブジェクト x の永続化を開始する前に，後者のスレッドが書き込みを開始すると，Race 2 が発生する．逆に，前者のスレッドが永続化を開始した後に，後者のスレッドが書き込みを開始すると，Race 3 が発生する．

さらに，別のスレッドは以下に示すようなプログラムを実行することも可能である．v は，u と同様に永続ルートの一つである．

```
v = x;
```

このスレッドは，オブジェクト x の複製を作成して永続化する．この場合，Race 1 が発生する可能性がある．

以降の節では，RBP を Total Store Ordering（TSO）メモリ一貫性モデルの下で動作させることを前提に，競合を解決する方法について述べる．Race 1 は 4.4.2 節および 4.4.3 節で解決する．Race 2 と Race 3 は，各々4.4.4 節と 4.4.5 節で解決する．

4.4.2 競合するスレッド間の依存関係

make_persistent は，オブジェクト r を開始点として**青オブジェクト**以外への参照のみを辿ることで，全ての**青オブジェクト**以外に到達できるようにする．このような**青オブジェクト**以外の集合を，make_persistent を呼び出すスレッドの**永続化対象集合**と呼ぶ．複数のスレッドが make_persistent を呼び出すと，それらの**永続化対象集合**が重複する可能性がある．この重複関係の推移閉包は同値関係である．この関係下の同値類を**干渉スレッドグループ**と呼ぶ．

例えば，図 4.5 に示すオブジェクトについて考える．スレッド S とスレッド T が，各々

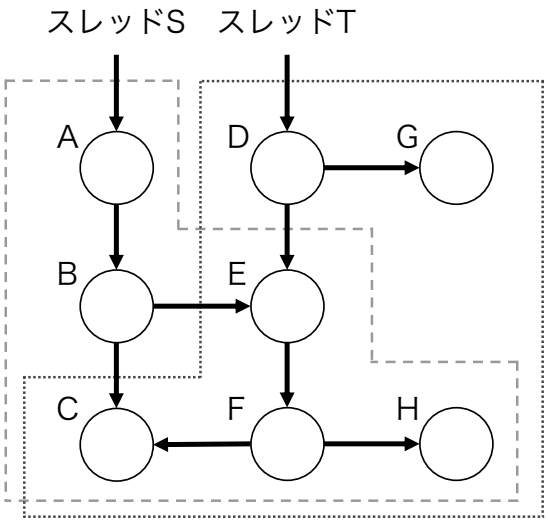


図 4.5　二つの永続化対象集合

オブジェクト A と D に対して make_persistent を呼び出すと，スレッド S の永続化対象集合は {A, B, C, E, F, H}，スレッド T の永続化対象集合は {C, D, E, F, G, H} となる．これら二つの集合は重複しているため，スレッド S とスレッド T は同じ干渉スレッドグループに属す.

複数のスレッドの永続化対象集合が重複するとき，これらのスレッドが同じオブジェクトを永続化して Race 1 が発生する可能性がある．例えば，複数のスレッドが同じオブジェクト o を同時に永続化することを考える．この競合を解決するために，オブジェクト o を**白オブジェクト**から**灰オブジェクト**に変化させたスレッドを，オブジェクト o の**担当スレッド**と定義する．この**担当スレッド**が，オブジェクト o を**黒オブジェクト**にする処理も担当する．

図 4.6 は，make_persistent から呼び出される shade のマルチスレッド版の擬似コードである．オブジェクト o が**白オブジェクト**である場合，nvm copy への転送ポインタをオブジェクト o の replica フィールドに書き込むことで，shade はオブジェクト o を**灰オブジェクト**にする．オブジェクト o が**白オブジェクト**であるかどうかは，オブジェクト o の replica フィールドが NULL であるかどうかを調べることで判断できる．別のスレッドが同時に同じオブジェクトを**灰オブジェクト**にする可能性があるため，この “**白オブジェクト**であるかどうかの確認（replica フィールドからの読み出しと比較）と，転送ポインタの書き

```
1   // オブジェクト o が白オブジェクトのとき，
2   // オブジェクト o を灰オブジェクトにして TRUE を返す
3   shade(DCopy o) {
4     NCopy on = alloc_NVM(o.size());
5     NCopy fwd = CAS(&o.replica, NULL, on);
6     if (fwd != NULL) {
7       // CAS に失敗した場合，オブジェクト o は灰，黒，青のいずれか
8       t = responsible_thread(fwd);
9       if (t != NULL && t != current_thread) {
10        // スレッド t はオブジェクト o を永続化する
11        depends_on(t);
12      }
13      return FALSE;
14    }
15    return TRUE;
16  }
```

図 4.6 マルチスレッド版 shade

込み（replica フィールドへの書き込み）” 処理は，アトミックに実行する必要がある．これには，*compare-and-swap* 命令を用いる．この命令は，図中で CAS（$addr$,v_1,v_2）のように抽象化されており，$addr$ はアクセスするアドレス，v_1 は現在 $addr$ に書き込まれていることが期待される値，v_2 は書き込む値である．CAS は，$addr$ の現在の値と v_1 の値を比較する．等しい場合は $addr$ に v_2 の値を書き込み，異なる場合は何もしない．等しいかどうかに関わらず，CAS は $addr$ の古い値を返す．つまり，CAS が返す値と v_1 の値が等しい場合，CAS が成功したことを示す．shade では，CAS（5 行目）がオブジェクト o の replica フィールドに，4 行目で割り当てられた nvm copy 領域への転送ポインタを書き込む．これは，オブジェクト o の状態が CAS を成功させたスレッドによって**灰オブジェクト**に変更されることを意味する．したがって，そのスレッドがオブジェクトの担当スレッドとなり，shade は TRUE を返す（15 行目）．

CAS に失敗した場合（7 行目），以下の 3 つの可能性が考えられる．

1 つ目が，オブジェクト o が既に現在のスレッドによって**灰オブジェクト**されている可能性である．つまり，オブジェクト o の担当スレッドは現在のスレッドである．これは，循環

参照や同じオブジェクトに対する複数の参照がある場合に発生する．この場合，特に必要な処理はない（13 行目）．

2 つ目が，オブジェクト o が別のスレッド t によって既に**灰オブジェクト**または**黒オブジェクト**にされている可能性である．つまり，オブジェクト o の担当スレッドはスレッド t である．この場合，depends_on（11 行目）で，現在のスレッドとスレッド t が同じ干渉スレッドグループに属すことを記録する．

3 つ目が，オブジェクト o が既にあるスレッドによって**青オブジェクト**にされている可能性である．この場合，特に必要な処理はない（13 行目）．

responsible_thread 関数は，**灰オブジェクト**または**黒オブジェクト**の担当スレッドを返す．**青オブジェクト**の場合，responsible_thread 関数は NULL を返す．

4.4.3　デッドロックの回避

スレッド T が別のスレッド S に依存している場合，不変条件 2 を維持するために，依存スレッド T は自身の永続化対象集合のオブジェクトを**青オブジェクト**にする前に，スレッド S の永続化対象集合が**青オブジェクト**になるのを待たなければならない．しかし，同じ干渉スレッドグループ内の複数のスレッドが相互的に依存する場合も考えられる．例えば，図 4.5 のオブジェクト C とオブジェクト E の担当スレッドを，各々スレッド S とスレッド T とする．スレッド S はオブジェクト E のためにスレッド T に依存し，スレッド T はオブジェクト C のためにスレッド S に依存するため，スレッド S とスレッド T が互いの処理を待つことになり，デッドロックが発生する．

デッドロックを回避するために，同じ干渉スレッドグループに属す全てのスレッドは，そのスレッドが担当する全てのオブジェクトを**黒オブジェクト**にした後，**バリア同期**を実行する．図 4.7 は，バリア同期を実行するマルチスレッド版の make_persistent の擬似コードである．シングルスレッド版との違いは，以下の通りである．

- LOCK と UNLOCK を呼び出す処理（詳細は 4.4.7 節）

- #が付いている行の処理（詳細は 4.4.5 節）
- SYNC を呼び出す処理

26 行目では，現在のスレッドが担当スレッドになっているオブジェクトが，全て**黒オブジェクト**になっている．次に，SFENCE 命令を発行して（28 行目）そのスレッドが担当する全てのオブジェクトの nvm copy のキャッシュラインを書き戻し，バリア同期を実行できる状態にする．その後，SYNC を呼び出して（31 行目）バリア同期を実行する．バリア同期が完了した時点で，干渉スレッドグループ内の全てのスレッドの永続化対象集合が同時に**青オブジェクト**になる．

4.4.4　白オブジェクトへの書き込み

スレッド T が putfield によって**白オブジェクト**のフィールドに値を書き込んでいる途中で，別のスレッド S が make_persistent によってそのオブジェクトの状態を**灰**，**黒**，**青**のいずれかに変化させる可能性がある．これは，Race 2 が発生する状況である．スレッド T は書き込み先を**白オブジェクト**として扱うため，dram copy のフィールドにのみ値を書き込む．その結果，不変条件 3 で保証している dram copy と nvm copy の整合性が壊れる．

図 4.8 は，マルチスレッド版の参照値の書き込みを行う putfield の擬似コードである．初めに，putfield は無条件にオブジェクト v への参照値を dram copy に書き込み（3 行目），replica フィールドに nvm copy への転送ポインタが書き込まれているかを確認する（5–6 行目）．

dram copy への書き込みと replica フィールドからの読み出しの実行順序を保証するために，MFENCE 命令を発行する（4 行目）．replica フィールドに転送ポインタが書き込まれていない場合（7 行目），書き込み先は**白オブジェクト**であるため，特に必要な処理はない．replica フィールドに転送ポインタが書き込まれている場合（10 行目），オブジェクトは**灰**，**黒**，**青**のいずれかであるため，putfield は挿入バリアとダブルアップデートバリアを実行する（11–13 行目）．最後に，CLWB 命令と SFENCE 命令（14–15 行目）を発行して，

```
1   // オブジェクト r を永続化する（つまり，青色にする）
2   make_persistent(DCopy r) {
3     if (shade(r) == TRUE) {  // オブジェクト r を白色から灰色にする
4       worklist.add(r);
5       while (not worklist.empty()) {
6         DCopy o = worklist.remove();
7         NCopy on = o.replica;
8         // オブジェクト o は灰色
9         LOCK(o);  // この LOCK の詳細は 4.4.7 節で述べる
10  #     RETRY: // # が付いている行の詳細は 4.4.5 節で述べる
11        // コピーステップ
12        foreach (Field f in o.fields) {
13          DCopy p = o[f];
14          if (shade(p) == TRUE) worklist.add(p);
15          on[f] = p.replica;
16        }
17  #     MFENCE;
18        CLWB_RANGE(on, on.size());
19  #     // 検証ステップ
20  #     foreach (Field f in o.fields)
21  #       if (o[f].replica != on[f]) goto RETRY;
22        // コピー処理完了
23        // オブジェクト o は黒色
24        UNLOCK(o);  // この UNLOCK の詳細は 4.4.7 節で述べる
25      }
26      // 現在のスレッドが担当スレッドになっているオブジェクトは，
27      // 全て黒オブジェクトになっている
28      SFENCE;
29      // 他のスレッドが担当スレッドになっているオブジェクトの色は不明
30    }
31    SYNC();
32    // オブジェクト r の推移閉包は青オブジェクト
33  }
```

図 4.7　マルチスレッド版 make_persistent

```
1   // オブジェクト o の参照型フィールド f に，オブジェクト v への参照を書き込む
2   putfield(DCopy o, Field f, DCopy v) {
3     o[f] = v;
4     MFENCE;
5     NCopy on = o.replica;
6     if (on == NULL) {
7       // オブジェクト o は白
8       return;
9     }
10    // オブジェクト o は灰，黒，青のいずれか
11    make_persistent(v);
12    NCopy vn = v.replica;
13    on[f] = vn;
14    CLWB(&on[f]);  // on[f] のキャッシュラインを書き戻す
15    SFENCE;
16  }
```

図 4.8 マルチスレッド版 putfield

nvm copy のフィールド f のキャッシュラインを書き戻す.

4.4.5 並行コピーにおける競合の解決

スレッド T がオブジェクト o に対して make_persistent を実行し，オブジェクト o の dram copy のフィールドをそのオブジェクトの nvm copy（擬似コード中の on）にコピーすることを考える．また，別のスレッド S が putfield を実行しており，スレッド T のコピー処理と同時にオブジェクト o のフィールド f に値 v を書き込むとする．以下のようなインターリーブは，Race 3 が発生する状況である．スレッド S によって NVM 上の on[f] に書き込まれた値は，スレッド T が上書きすることで失われる．その結果，不変条件 3 で保証している dram copy と nvm copy の整合性が壊れる.

step	スレッド T	スレッド S
1:	`p = o[f]`	
2:		`o[f] = v`
3:		`on[f] = v.replica`
4:	`on[f] = p.replica`	

この競合を解決するために，データレースフリーを前提とした TSO メモリ一貫性モデルのための並行コピー GC である Transactional Sapphire GC [18] のコピープロトコルを用いる．図 4.7 の “#” が付いている行は，このプロトコルの処理である．

このコピープロトコルは，コピーステップ（12–16 行目）と検証ステップ（20–21 行目）で構成されている．まず，競合の有無に関わらず，単純にフィールドのデータをコピーする．次に，コピーの結果を検証する．つまり，`make_persistent` のコピーステップで読み出した dram copy の値と同じ値を，nvm copy が持っているか検証する．フィールド `f` のためのこの確認は，フィールド `f` から参照されるオブジェクトの nvm copy （`o[f].replica`）と，オブジェクト `o` の nvm copy のフィールド `f` から参照される nvm copy （`on[f]`）を比較することで行われる（21 行目）．検証ステップに失敗した場合は，コピーステップを再実行する（21 行目）．17 行目の `MFENCE` 命令は，コピーステップ中の他のスレッドによる書き込みと，検証ステップの読み出しが並べ替えられないことを保証するために挿入する．

4.4.6 補助関数

RBP は `responsible_thread`，`depends_on`，`SYNC` の 3 つの補助関数を使用する．本節では，これらの関数に課す要件と，実装上の問題について説明する．

`responsible_thread` 関数は，オブジェクト `o` の nvm copy を引数に与える．この関数の要件は，オブジェクト `o` が**灰オブジェクト**または**黒オブジェクト**の場合に，常に担当スレッドを返すことである．オブジェクト `o` が**青オブジェクト**である場合は，`NULL` 等の特別な値を返す必要がある．ただし，戻り値を `depends_on` 関数が正しく処理できる場合に限

り，この特別な値を返す処理は不要である点に注意されたい．

本研究では，全ての nvm copy に担当スレッドへの参照をもたせることで，responsible_thread 関数の機能を実現した．この参照は，nvm copy を割り当てた際にセットされ，**青オブジェクト**になる際にクリアされる．alloc_NVM は，各 nvm copy に 2 ワードのフィールドを追加して割り当てる．この追加フィールドは，nvm copy のヘッダである thread フィールドと next フィールドに利用する．前者は担当スレッドを保持するために，後者は同じ担当スレッドである nvm copy 同士でリストを構築するために使用される．make_persistent を実行するスレッドは，このリストの先頭を保持する．make_persistent の SYNC を実行すると，このスレッドが永続化を担当した全てのオブジェクトは全て**青オブジェクト**になる．このとき，リスト内の全ての nvm copy の thread フィールドをクリアし，responsible_thread 関数が NULL を返すようにする．SYNC の完了と thread フィールドのクリアには時間差があるため，**青オブジェクト**の thread フィールドがまだクリアされていない可能性がある点に注意されたい．

depends_on 関数は，引数に responsible_thread 関数が返す依存先のスレッド t を与える．この関数は，現在のスレッドと依存先のスレッド t が属す干渉スレッドグループを統合する．本研究では，互いに素な集合群を管理するデータ構造である**素集合データ構造**（*disjoint-set data structure*）と，それを操作する *Union-Find* **アルゴリズム**によって，スレッド間の依存関係を管理する．Union-Find アルゴリズムは，2 つの集合を統合する処理（Union）と，ある要素がどの集合に属すかを調べる処理（Find）をサポートする．集合に属す要素の管理には，木構造を用いる．2 つの集合を統合する処理（Union）は，一方の集合の根ノードが，もう一方の集合の根ノードの子となることで実現する．これを，複数のスレッドが同時に呼び出しても問題にならないよう，適切に排他制御の処理を挿入して実装する．

SYNC 関数は，同じ干渉スレッドグループに属す全てのスレッド間でバリア同期を実行する．SYNC 関数を呼び出したスレッドは，同じ干渉スレッドグループ内の全てのスレッドが SYNC を呼び出すまで待機する．全てのスレッドが揃い，バリア同期が完了した後に，干渉ス

レッドグループの依存関係をリセットしなければならない．本研究では，先述の Union-Find アルゴリズムに加えて，“ある要素を集合から取り除き，その要素だけの集合を作成する機能” を実装した．

スレッド t が SYNC 関数の実行を完了した後に，別のスレッドが depends_on 関数を呼び出してスレッド t を干渉スレッドグループに統合しないよう注意が必要である．統合した場合，その干渉スレッドグループに属す全てのスレッドはスレッド t を永久に待機する可能性がある．しかし，バリア同期と thread フィールドのクリアには時間差があるため，これを完全に防ぐことはできない．先述の通り，depends_on 関数で干渉スレッドグループを統合する際に Union で操作するのは親ノードである．本研究では，集合から要素を取り除く操作を葉ノードに位置するノードから順に行うことで，永久に処理されない親ノードを作り出さないようにする緩和策を講じた．

4.4.7 データレースフリーの例外

RBP で動作させる Java プログラムはデータレースフリーであることを仮定しているが，実用上の理由より二つの例外を設けている．一つは，java.util.concurrent.atomic パッケージによるアトミックな読み書き操作である．この操作は排他制御なしにフィールドへアクセスする目的で提供されており，フィールドへの競合アクセスを禁止して排他制御を強制するのは不便である．もう一つは，volatile フィールドへの読み書き操作である．Java の volatile フィールドは，複数のスレッドによって同時にアクセスすることが想定されている．例えば，生産者スレッドが何らかのデータを生成した後に，フラグを volatile フィールドにセットし，消費者スレッドがフラグをポーリングしてフラグがセットされたことを確認するような，同期の用途に使われる．したがって，volatile フィールドへの競合アクセスを禁止するのは不便である．

上記の二つの状況に対処するために，アトミックな読み書きと volatile フィールドへの読み書きで競合アクセスを可能にする特別なライトバリアを提供する．アトミックな読み書きについては，スピンロックを用いたオブジェクト単位の排他制御によって解決する．これ

は，後述する LOCK と UNLOCK によって実現される．以降は，volatile フィールドへの読み書きに関する解決策を述べる．

図 4.8 に示す putfield 関数のライトバリアは，putfield が永続化中のオブジェクトに書き込みを試みた場合でも，make_persistent による永続化処理が正しく実行されるように設計されている．具体的には，4.4.4 節で述べたように，まず putfield は dram copy に書き込みを行い，その後 nvm copy に書き込みを行う．これにより，make_persistent はコピーステップの途中で書き込みが行われたことを検証ステップで検出し，コピーステップを再実行することができる．

しかし，あるスレッドが volatile フィールドに書き込みを行っている間に，別のスレッドがそのフィールドの値を読み出している場合，putfield で nvm copy よりも先に dram copy に書き込みを行うと問題が発生する．これは，putfield が dram copy に書き込んで nvm copy にはまだ書き込んでいない値を，別のスレッドが読み出す可能性があるためである．値を読み出したスレッドが，その値に依存する書き込みを実行する場合，その書き込みは依存元（読み出した volatile フィールド）の値が永続化される前に永続化される可能性がある．この時点でシステムがクラッシュした場合，NVM はクラッシュのない通常の実行では発生しない不整合な状態になる．

この問題を解決するために，volatile フィールドへの書き込み時は永続化処理との並行動作を諦め，まず nvm copy に書き込みを行い，その後 dram copy に書き込みを行う．この順序で書き込みを行えば，不整合な状態にはならない．仮に，あるスレッドが volatile 用の putfield で nvm copy に書き込んで dram copy にはまだ書き込んでいない値を，別のスレッドが dram copy から読み出しても，その結果は “先に volatile フィールドの読み出しが発生して，その後に書き込みが発生した状況” と同等である．

書き込み順序の変更に加えて，次の二つの目的でスピンロックによる排他制御を行う．一つは，あるスレッドが volatile フィールドに書き込みを行っている間に，別のスレッドがそのオブジェクトを複製を作成して永続化するのを防ぐためである．もう一つが，複数のスレッドが同じ volatile フィールドに同時に書き込みを行うことを防ぐためである．

```
1  // オブジェクト v への参照を，オブジェクト o の volatile フィールド f に書き込む
2  putfield_volatile(DCopy o, Field f, DCopy v) {
3    LOCK(o);
4    NCopy on = o.replica;
5    if (on != NULL) {
6      // オブジェクト o は灰，黒，青のいずれか
7      make_persistent(v);
8      NCopy vn = v.replica;
9      on[f] = vn;
10     CLWB(&on[f]);  // on[f] のキャッシュラインを書き戻す
11     SFENCE;
12   }
13   o[f] = v;
14   UNLOCK(o);
15 }
```

図 4.9 volatile フィールド版 putfield

図 4.9 は，volatile フィールド版の putfield の擬似コードである．書き込み先のオブジェクト o が nvm copy を持つ場合，まず nvm copy に書き込みを行い（9 行目），次に dram copy に書き込みを行う（13 行目）．LOCK（3 行目）と UNLOCK（14 行目）は，永続化処理と volatile フィールドへの書き込み，または複数スレッドによる同じ volatile フィールドへのアクセスを排他制御するためのものである．また，図 4.7 の make_persistent の 9 行目と 24 行目にも，LOCK と UNLOCK が必要である．なお，make_persistent の LOCK と UNLOCK では，通常のフィールド（volatile 以外のフィールド）への書き込み処理との排他制御は行えず，4.4.5 節で示した検証ステップによる競合検出は依然として必要である．

LOCK と UNLOCK による排他制御は，**オブジェクト単位**で行う．つまり，処理しているオブジェクトが同じ場合は排他的に動作し，異なる場合は並行動作する．これを実現するために，全ての dram copy のヘッダにロック用のビットを用意する必要がある．このビットには，dram copy の replica フィールドの最下位ビット（LSB）を用いる．nvm copy はワードサイズ（8 バイト）でアライメントされているため，nvm copy を示すアドレスの下位 3 ビットは常に 0 の状態である．ロック用ビットの書き換えは，常に CAS を用いて行われる．

以上のような volatile フィールドに関するアルゴリズムの変更は，通常のフィールドの読み書きには影響を与えない．また，volatile フィールドからの読み出しにも一切関与しないため，リードバリアは依然として不要である．

4.4.8 競合するアクセスの結果

RBP で動作させる Java プログラムは，4.4.7 節で述べた例外を除いてデータレースフリーであると仮定している．それにも関わらず競合するアクセスを行うと，以下のような状況になることが考えられる．

読み出しと書き込みの競合が発生すると，永続化の順序が正しくない可能性がある．スレッド T による書き込み W_1 と，スレッド S による読み出しが同じフィールドに対して行われる場合，スレッド S は永続化前の CPU キャッシュメモリから W_1 の値を読み出す可能性がある．スレッド S が読み出した値に応じて別の書き込み W_2 を実行すると，W_2 が W_1 の前に永続化され，依存関係に矛盾が生じる可能性がある．

複数の書き込みが競合すると，誤った値を永続化する可能性がある．二つの書き込み W_1 と W_2 が同じフィールドに対して同時に実行される場合，W_1 の値は NVM に永続化され，W_2 の値は DRAM に存在すること等が考えられる．

4.5 NVM のメモリ管理

本研究で利用する NVM 専用のメモリアロケータおよび GC 等のメモリ管理機構は，本研究と並行して長安らにより開発された [9]．本節では，その長安らのメモリ管理機構について説明する．

NVM 領域の管理機構は，Java スレッドから要求された nvm copy 用の NVM 領域の確保や，GC による不要な nvm copy の回収を行う．本管理機構は， nvm copy への読み書きをできる限り行わないことを原則とする．これにより，管理機構に関わるオーバヘッドを減らすと同時に，GC 中のクラッシュによって nvm copy の整合性が失われることを防ぐ．

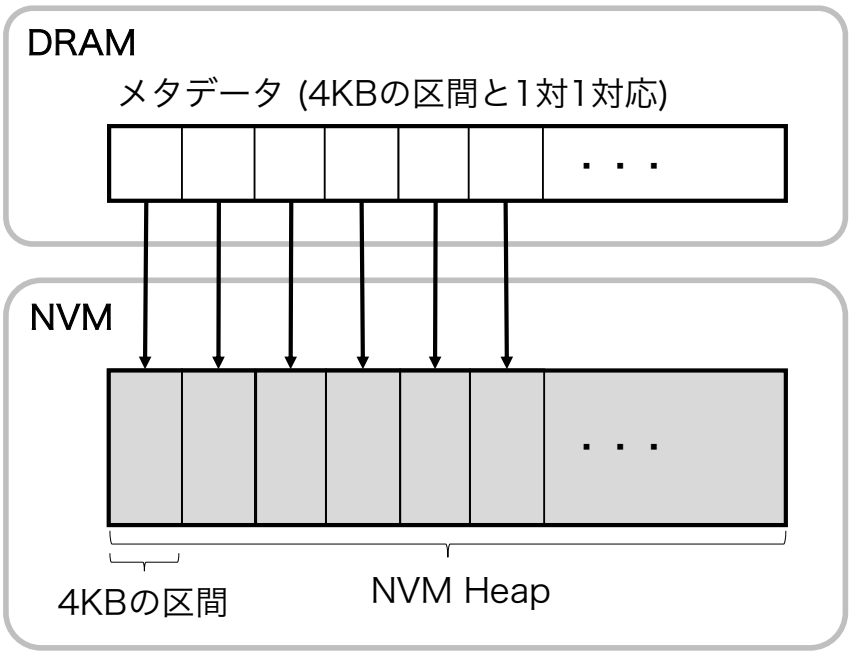


図 4.10　NVM 領域の管理機構

4.5.1　NVM アロケータ

NVM アロケータはオブジェクトに，予め決めた 40 段階のサイズに切り上げた領域を割り当てる．この 40 段階のサイズを，サイズクラスという．

図 4.10 は，NVM アロケータの概略である．NVM 上のヒープ（5.1 節で説明する NVM Heap）は 4KB ごとの区間に区切り，各区間には同じサイズクラスのオブジェクトだけを配置する．各区間に対して 1 つ，メモリ管理用のメタデータを DRAM 上に作成し，Java スレッドはこれらを通じて NVM 領域の確保と解放を行う．メタデータはアロケーションの状況を記憶する領域や，GC 時のマーク用領域などを持つ．これらの情報は DRAM 上にあるのでクラッシュによって失われるが，5.3.3 節で述べる復旧手順により NVM 上にあるデータからリカバリが可能であるため，整合性は保たれる．

図 4.11 は，サイズクラスが 24 バイトの区間とメタデータの概略である．区間は，NVM 上の 4KB の領域を指す．この区間のサイズクラスは 24 バイトであるため，24 バイトのオブジェクトのみを割り当てる．DRAM 上のメタデータは，サイズクラスとアロケーションの状況を記憶する領域を示している．アロケーション状況は，区間の左から順に使用中を 1,

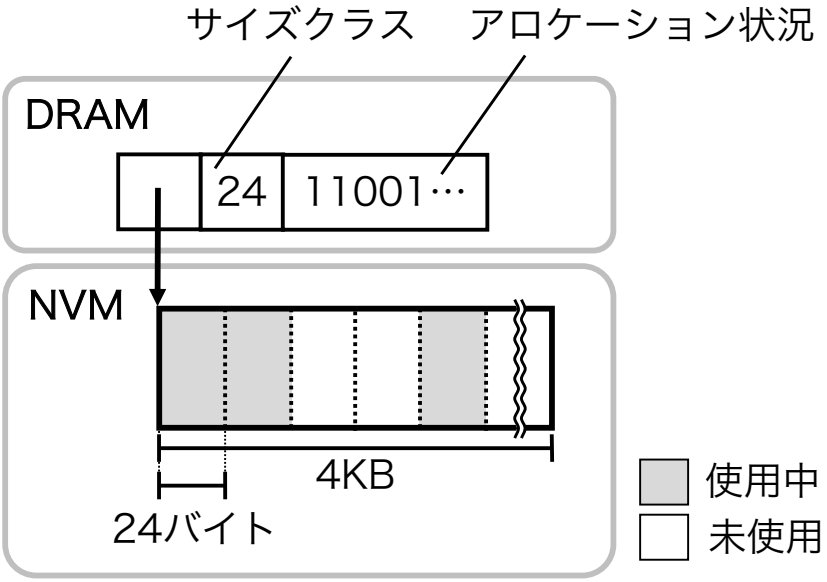


図 4.11　サイズクラスが 24 バイトの区間とメタデータ

未使用を 0 としてビット列で表現する．この図では，GC 時のマーク用領域を省略している．

Java スレッドは，各サイズクラスに対応した NVM 領域の区間と DRAM 上のメタデータを，スレッドローカルに保持する．これにより，スレッド間の競合を避ける．

大きなサイズのオブジェクトの NVM 領域は，区間を使わずフリーリストで管理する．このときメタデータは各 nvm copy のヘッダとして，直接 NVM 上に配置する．したがって，大きなオブジェクトのアクセスにはスレッド間の競合が発生し得るため排他制御が必要となるが，その割当回数は全体からすると稀であると期待できる．

4.5.2　NVM 領域の GC

nvm copy は，読み書きを極力減らすためにマーク&スイープ方式で GC を実行する．

GC に nvm copy の移動を伴うと，移動中のクラッシュにより整合性が失われないようにする必要があり，オーバヘッドとなり得る．マーク&スイープ方式の GC ではオブジェクトを移動させないため，整合性が失われることはない．

第 5 章

リカバリ機構

本章では，リカバリ機構を実現するための準備である永続化機構の改造と，3.2 節で示したリカバリ API である `recovery` メソッドの詳細な設計について記す．

5.1　永続化機構の改造

図 5.1 は，OpenJDK と永続化機構が作成したデータの概略である．斜線柄の四角形はクラスオブジェクトを，ドット柄の四角形はクラス情報を保持するデータ Klass を示す．nvm copy のみではリカバリが実現できないため，永続化機構の改造を行う．

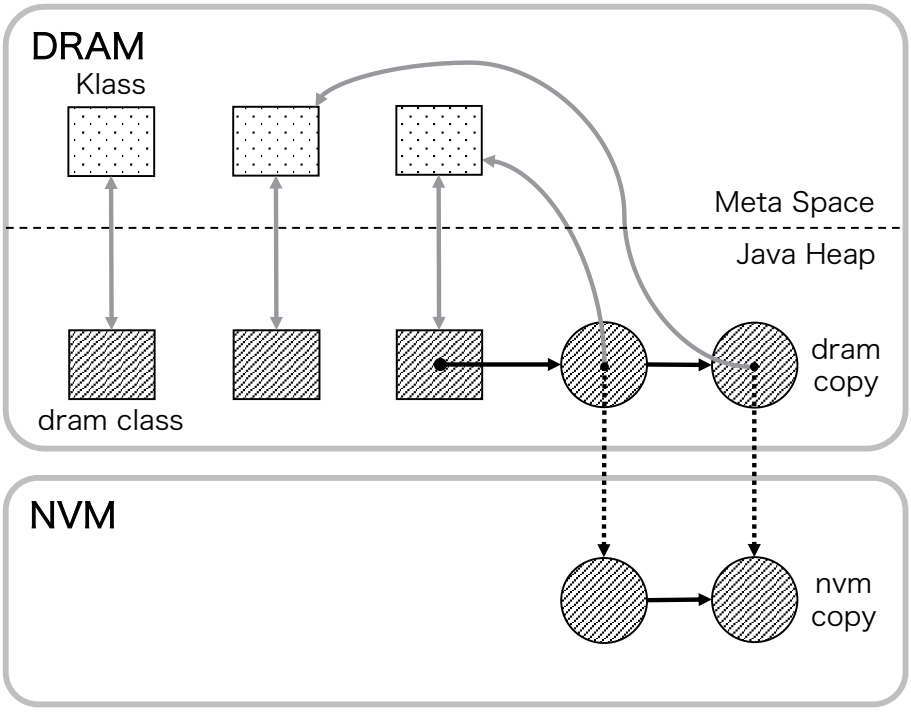


図 5.1　OpenJDK と永続化機構が作成するデータ

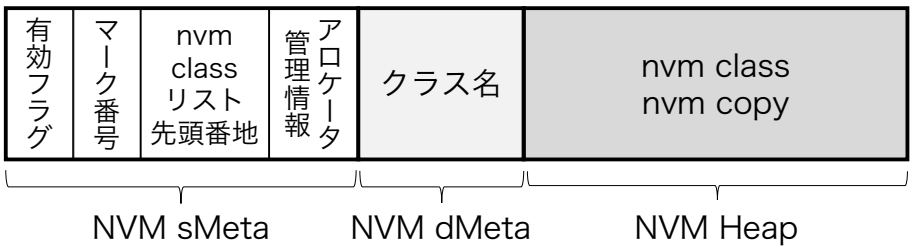


図 5.2　NVM 領域の内訳

リカバリ処理に必要なデータを NVM に置くために，NVM 領域を以下の 3 つに分割する．

- NVM sMeta: メタ情報を置く静的な領域
- NVM dMeta: メタ情報を置く動的な領域
- NVM Heap: nvm copy を置く動的な領域

図 5.2 に，NVM 領域の内訳を表す．NVM sMeta 領域は，メタ情報を記録するための静的な領域である．アドレスは固定であり，同じプログラムならば，どの実行でも同じアドレスにデータを記録する．NVM dMeta 領域には，メタ情報の中でも動的に領域を割り当てるものを記録する．NVM Heap 領域は，nvm copy を配置する動的な領域である．この領域は，4.5 節で示した NVM アロケータによって管理する．

5.1.1　永続ルートの永続化

永続ルートは static フィールドの一部である．OpenJDK 16 では static フィールドはクラスオブジェクトの末尾に記録されている．RBP では，全ての DRAM 上のクラスオブジェクトの複製を NVM に置いて永続化することで，永続ルートの永続化を実現した．ただし，永続ルート以外の static フィールドは全て永続化の対象外とする．以後，2.3.3 節で述べたように他のオブジェクトと区別するため，DRAM 上のクラスオブジェクトを dram class，NVM 上のクラスオブジェクトを nvm class と表記する．

図 5.3 に，OpenJDK と改造した永続化機構が作成するデータを示す．nvm class の未使用領域（永続ルート以外のフィールド領域）を利用して，nvm class の連結リストを構築す

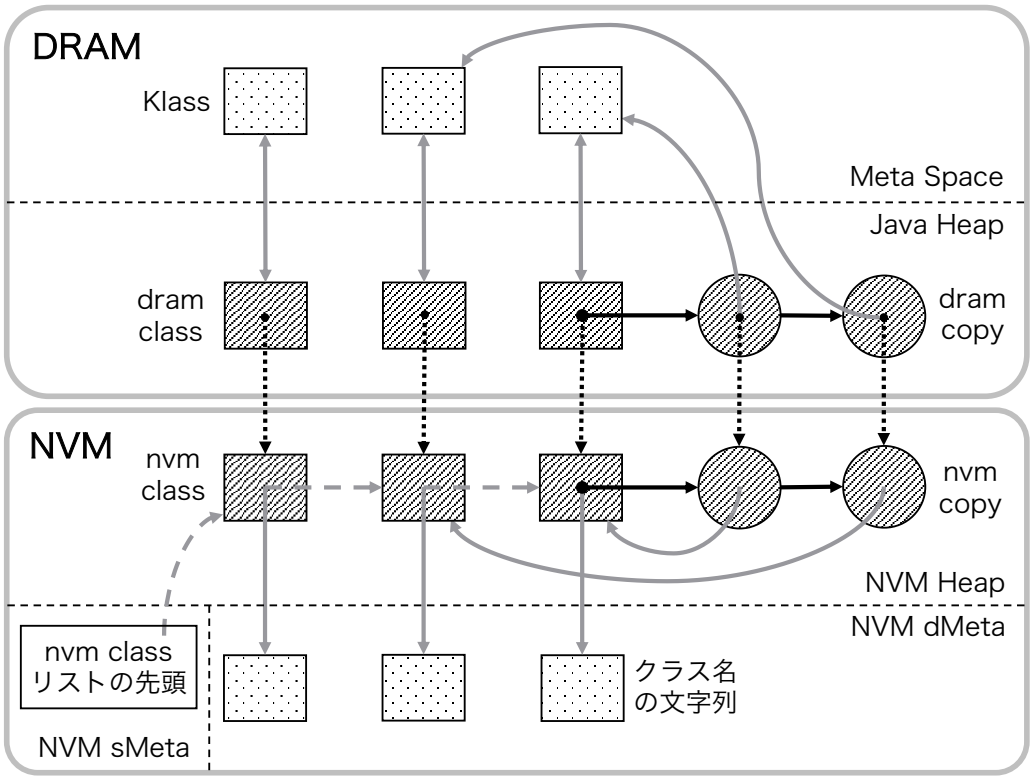


図 5.3　OpenJDK と改造した永続化機構が作成するデータ

る．リストの先頭を指す参照は，NVM sMeta 領域に記録する．NVM 上にある左下の四角形はリストの先頭アドレスを，灰色の点線矢印は連結リストの参照を示している．NVM 上にある斜線柄の四角形は，nvm class を示している．nvm copy と同様に nvm class は NVM Heap 領域に配置する．また，dram copy と同様に dram class のヘッダにも nvm class への転送ポインタを持たせる．クラスオブジェクトへの書き込みは，永続ルートであるフィールドに限り dram class と nvm class の両方に対して行う．

5.1.2　クラス名の永続化

dram copy は，ヘッダに Klass への参照を保持することで，そのオブジェクトが属す Java クラスを把握している．これに対して，nvm copy では Klass の代わりに nvm class への参照を持たせる．さらに，クラス名の文字列を永続化して nvm class の未使用領域からそれへの参照を保持する．Klass は Java クラスをロードすることで再生成されるため，それを識

別する最低限の情報であるクラス名の文字列が永続化されていれば良い．文字列を配置する領域は，NVM dMeta 領域から動的に確保する．

図 5.3 の NVM 上にあるドット柄の四角形はクラス名の文字列を，灰色の実線矢印はクラス名の文字列への参照を示している．

5.2　処理の流れと目標

リカバリ処理では，メモリがクラッシュ前の図 5.3 の状態になるよう，NVM 上のデータを元に DRAM 上にデータを再構築する．リカバリ処理の主な流れは，以下の通りである．

1. 永続化されているクラス名を探索してクラスをロードする．
2. nvm copy を永続ルートから探索して，DRAM 上に全ての永続オブジェクトの dram copy 領域を確保する．
3. NVM アロケータの管理情報を復元する．
4. nvm copy のヘッダに dram copy への転送ポインタをセットして，参照値変換の準備を行う．
5. nvm copy から dram copy にフィールドのデータをコピーする．

この処理を，以下の目標を満たすように注意して設計する．

1. GC によるオブジェクトの回収と移動に対処する．
2. 永続ルートと永続オブジェクトは，Java プログラムから見てアトミックに復元される．

目標 1 の GC は，リカバリ処理の dram copy 領域の確保または他のアプリケーションスレッドの実行によって，常に発生する可能性がある．GC が発生すると，dram copy 領域が回収される危険性がある．一方，メモリ不足の場合は一旦 GC を動作させて空き領域を作る必要がある．そのため，dram copy 領域を確保している間は GC を停止させる解決策が取れず，他の対処が必要である．また，リカバリ処理では nvm copy のヘッダに dram copy

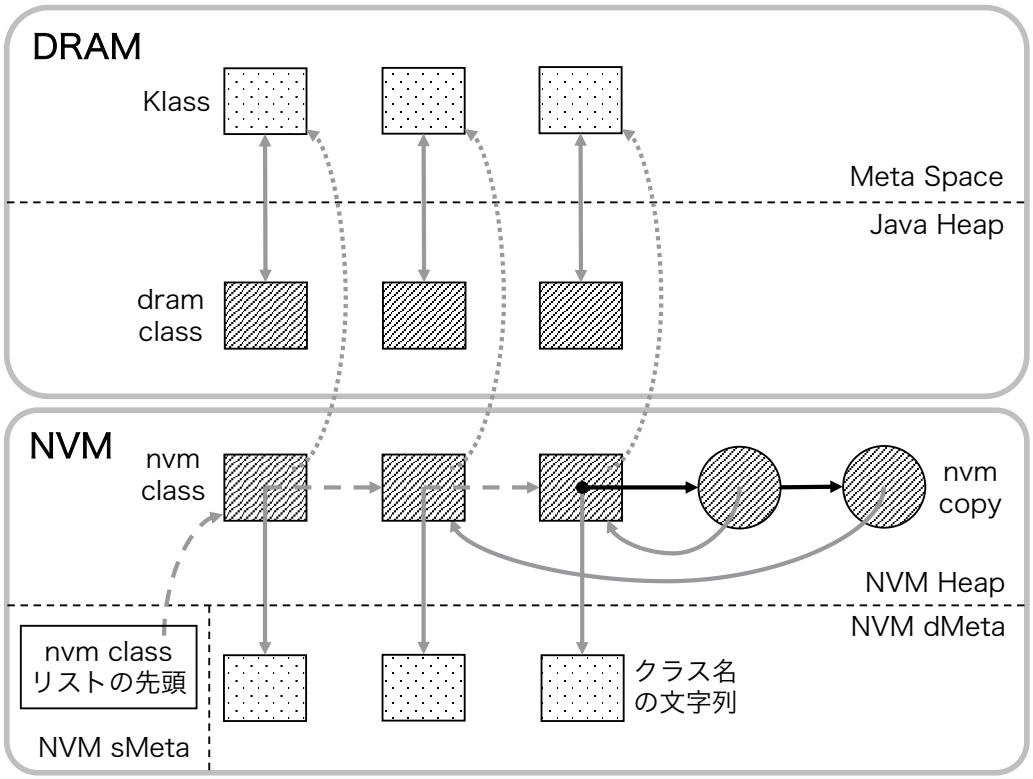


図 5.4　クラスのロード後のメモリの状態

への転送ポインタをセットする．GC が dram copy を移動させると，nvm copy に書き込まれた参照は dangling pointer となるため，対処が必要である．

目標 2 は，一部の永続ルートや永続オブジェクトだけにアクセス可能な状態で，Java プログラムが動作することはないことを意味する．

5.3　詳細設計

5.3.1　クラスのロード

リカバリ処理にはクラスの情報が必要であるため，永続化されたクラス名の Java クラスを全てロードする．永続化された全てのクラス名は，nvm class の連結リストを走査することで取得できる．Java クラスのロードには `java.lang.Class` クラスの `forName` メソッドを用いることで，複雑な Java クラスのロード処理を簡略化する．

図 5.4 に，クラスのロード後のメモリの状態を示す．クラスのロード処理によって，Klass と dram class が作成される．その後，nvm class のヘッダに Klass への参照をセットする．この参照によって，nvm copy のオブジェクトが属すクラスの詳細な情報を得ることができる．

Klass は DRAM 上の GC の対象ではない Meta Space 領域に配置されるためリカバリ中にアドレスが変わることはなく，nvm class から Klass への参照は dangling pointer にならない．また，この参照は NVM から DRAM のデータを指すことになるが，クラッシュ後の実行ではこれを再利用せずセットし直すため，dangling pointer になっても問題はない．

5.3.2 dram copy 領域の確保

nvm copy を永続ルートから探索して，DRAM 上に全ての永続オブジェクトの dram copy 領域を確保する．

永続ルートは nvm class に保持されているため，nvm class の連結リストを走査して全ての永続ルートを集める．次に，集めた永続ルートから参照を辿って到達可能な nvm copy を探索する．

重複した探索を防ぐため，探索済みであるかどうかをマークするフラグを nvm copy のヘッダに保持する．このヘッダを 0 で初期化しておき，探索したオブジェクトのフラグを 1 に書き換える．書き換える前にフラグが既に 1 であるオブジェクトは探索しない．

探索して見つけた各 nvm copy 毎に，DRAM 上に dram copy 領域を確保する．dram copy 領域の確保には，OpenJDK の既存のアロケータを利用する．dram copy の既存ヘッダを適切に初期化し，追加ヘッダである nvm copy への転送ポインタもセットする．フィールドのデータは `null` で初期化された状態である．

確保した dram copy の領域は，GC によって回収される可能性がある．しかし，メモリ不足の場合は一旦 GC を動作させて空き領域を作る必要があるため，GC を停止させることはできない．そこで，この問題は全ての dram copy を参照するオブジェクトを DRAM 上に作成することで解決する．図 5.5 の上部ある灰色の四角形は，参照型の配列オブジェクト

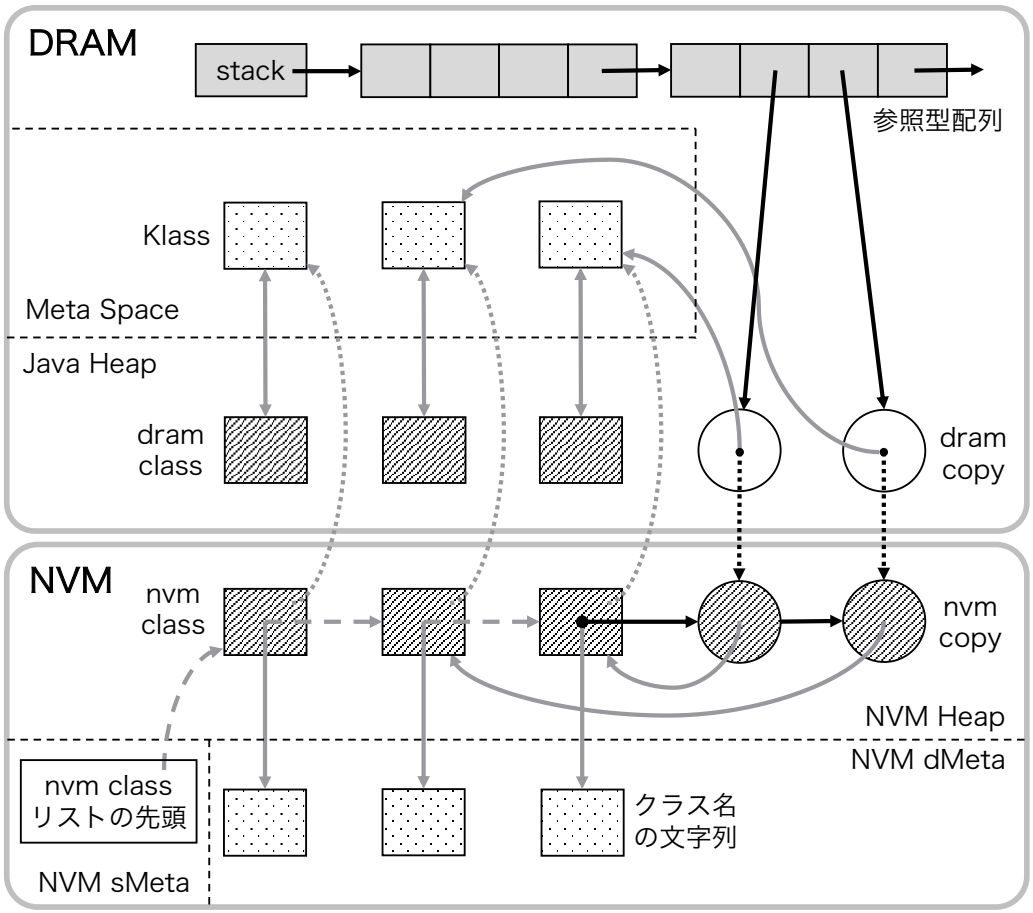


図 5.5　dram copy 領域を確保した後のメモリの状態

を示している．dram copy を確保した直後に，配列オブジェクトの末尾以外の要素に dram copy への参照を格納する．要素の空きが無くなると，新たに参照型の配列オブジェクトを作成して，古い配列オブジェクトの末尾に新しい配列オブジェクトへの参照を持たせる．配列オブジェクトを GC のルート集合に挿入する操作は煩雑であるため，これを Java プログラムで実装して簡略化した．具体的には，Java プログラムで 1 つ目の配列オブジェクトを作成して，それをローカル変数から参照した．

5.3.3　NVM アロケータのリカバリ

4.5 節で述べた NVM の管理情報は DRAM 上に配置されており，クラッシュによって消えてしまうためリカバリが必要である．NVM アロケータのリカバリでは，NVM の管理用

メタデータを作成し，そのアロケーション情報を正しく設定する．これには，以下の条件を必要とする．

1. クラス情報にアクセスできること．
2. 使用済み領域と未使用領域の境界アドレスが永続化されていること．

各区間のサイズクラスは，その区間に存在するオブジェクトのサイズから復元する．このオブジェクトのサイズを得るために，クラス情報が必要である．

NVM アロケータは，NVM Heap を前から 4KB ずつ区切って使用しており，使用済み領域と未使用領域の境界アドレスのみを永続化している．リカバリ中に NVM 領域が必要になったとき，使用済み領域から確保すると有効な nvm copy を再利用してしまう可能性がある．そこで，その場合は永続化された境界アドレス以降の未使用領域から確保する．

ここからは，リカバリの手順について説明する．まず，初期状態のメタデータを DRAM 上に作成する．ただし，この時点では各区間に対応するメタデータのアロケーション情報は記録されておらず，そのメタデータに対応する区間のサイズクラスも不明である．

次に，コピー元の nvm copy に対応するアロケーション情報をメタデータに保存する．これは，nvm copy のフィールドデータのコピーと同時に行う．また，その区間で最初に記録される nvm copy を処理する場合，その区間がアロケーションすべきサイズクラスを，そのオブジェクトのサイズと同じ値に設定する．全ての nvm copy のフィールドデータのコピーが終わったとき，メタデータのアロケーション情報のリカバリも完了する．

サイズの大きなオブジェクトの NVM 領域は区間を使わずフリーリストで管理しており，同様の手順でメタデータのリカバリが可能である．

5.3.4 参照値変換の準備

この処理から，他のスレッドを停止させた状態でリカバリ処理を進める．また，この処理以降でメモリの割り当ては行わない．そのため，GC は起こらない．

5.3.5 節で示すフィールドデータのコピー処理では，nvm copy への参照を dram copy へ

の参照に変換する必要がある．この変換のために，全ての nvm copy のヘッダに dram copy への転送ポインタを持たせる．

この処理の間は GC が起きないので，転送ポインタの先が移動することはない．また，nvm copy から dram copy への参照は NVM から DRAM のデータを指すことになるが，クラッシュ後の実行ではこれを再利用せずセットし直すため，DRAM 上のデータが消失しても問題はない．

5.3.5 フィールドデータのコピー

引き続き，この処理も他のスレッドを全て停止させて行われる．

nvm copy から dram copy にフィールドのデータをコピーする．dram copy への書き込みは，永続化機能を持たないライトバリアを用いて行う．nvm copy に格納されている参照値は nvm copy を指しているため，dram copy にコピーする場合は 5.3.4 節でセットしたデータを用いて dram copy を指す参照値に変換してから書き込む．クラスオブジェクトも同様に，永続ルートのフィールドのみ nvm class から dram class にデータをコピーする．

その後，dram class のヘッダに nvm class への転送ポインタをセットする．最後に，リカバリ処理のために nvm copy のヘッダに書き込んだ Klass と dram copy への参照をクリアする．

5.3.4 節で示した通り，本節の永続ルートと永続オブジェクトへの書き込みは他のスレッドを停止させた状態で行う．そのため，Java プログラムは本節の処理中に実行されず，実行前または実行後の状態のみ観測できる．つまり，全ての永続ルートと永続オブジェクトは Java プログラムから見てアトミックに復元される．

以上のリカバリ処理により，メモリをクラッシュ前の状態に戻すことができた．

5.4　リカバリ中のクラッシュへの対処

5.3.2 節で示した手順では，リカバリ中にクラッシュするとリカバリができなくなってしまう．この節では，リカバリ中にクラッシュしても再度リカバリできるように手順を変更する．

5.3.2 節の永続ルートから nvm copy を探索する処理は，クラッシュで問題になる唯一の処理である．この処理では，重複した探索を防ぐため，nvm copy のヘッダに探索済みであるかどうかをマークするフラグを保持しており，未探索であれば 0，探索済みであれば非 0 の値を書き込む．

しかし，nvm copy にマークを付けた状態でクラッシュが発生すると，次回のリカバリ処理では探索処理前にも関わらずフラグがセットされている nvm copy が存在することになる．フラグを 0 と 1 の 2 値とすると，nvm copy が探索済みであるかを正しく判別できない．この問題を解決するため，RBP では，マーク番号と呼ぶ符号なし 64 ビット整数をフラグに用いる．マーク番号は `recovery` メソッドを実行する毎に 1 ずつ加算され，1 から $2^{64} - 1$ の範囲で変化する．現在のマーク番号は NVM sMeta 領域に永続化されており，フラグが現在のマーク番号と同じであれば探索済み，異なれば未探索であると判別する．これにより，nvm copy の探索中に連続でクラッシュが発生しても，現実的な回数はリカバリ処理を再試行することが可能である．全ての nvm copy を探索し終えると，nvm copy のヘッダを 0 にリセットして，現在のマーク番号を 1 に戻す．

第 6 章

永続化の動作の検証

6.1 検証方法

二つの Java アプリケーションを作成し，提案システム上で故意にクラッシュさせ，正しく永続化とリカバリが動作することを確認した．アプリケーションは，生成フェーズと検証フェーズの二つで構成されている．まず，あるデータを作成して永続オブジェクトに格納していく生成フェーズを実行する．全てのデータが揃った後に，永続オブジェクトに格納されているデータが正しいものであるか確認する検証フェーズを実行する．

このアプリケーションの生成フェーズまたはリカバリ処理の途中で，システムをクラッシュさせる．何度かクラッシュとリカバリを繰り返す．その後の検証が成功すれば，正しく永続化とリカバリができていると見做す．システムのクラッシュは，計算機の電源を遮断する方法で 50 回，SIGKILL シグナルを送る方法で 3 万回発生させた．後者は高速に試行できるが，プログラム停止時に CPU キャッシュのデータを NVM に書き戻すため，クラッシュを完全には再現できていない点に留意する必要がある．

二つのアプリケーションとは，素数生成アプリケーション（6.2 節）と文字列生成アプリケーションであり，これらを用いて動作確認を行った．素数生成アプリケーションは，昇順に 1000 万個の素数を永続オブジェクトに格納する．生成フェーズの素数判定は，調べる数の平方根以下の素数で割る手法を用いる．これは，これまでに生成して永続化されている素数のデータを利用する．検証フェーズの素数判定は，調べる数の平方根以下の数で割る手法を用いる．これは，永続化したデータを利用しない．文字列生成アプリケーションは，文字列を 1000 万個生成して永続オブジェクトに格納する．n 番目の文字列は固定文字列と数字

n を結合して生成する．生成フェーズと検証フェーズの両方でこの手法を用いる．

6.2 Java アプリケーションの実装例

RBP の永続化機構およびリカバリ機構を用いた Java アプリケーションの実装例を示す．

図 6.1 と図 6.2 は，RBP を用いて素数の生成を行う Java プログラムである．`PrimeGenerator` クラスは，素数を生成するアプリケーションである．プログラムを起動すると，`Primes` クラスの `main` メソッドが実行され，`PrimeGenerator` クラスのメソッドを呼び出す．

6.2.1 `Primes` クラス

図 6.1 の `Primes` クラスについて説明する．

4 行目と 5 行目は，`durableroot` アノテーションを付けて `PrimeGenerator` 型の static フィールドを永続ルートとして定義している．

8 行目から 15 行目は，永続オブジェクトのリカバリまたは初期化に関するコードである．この処理の詳細は 3.2 節を参照されたい．

リカバリが行われた場合，`generator` フィールドには参照値が格納された状態となる．リカバリが行われなかった場合，`generator` フィールドには初期値の `null` が格納された状態となる．その際は，18 行目で新たに `PrimeGenerator` のオブジェクトを作成する．

その後，21 行目でアプリケーションを実行する．この処理の詳細は 6.2.2 節で示す．

6.2.2 `PrimeGenerator` クラス

図 6.2 の `PrimeGenerator` クラスについて説明する．

2 行目は生成した素数を格納する `int` 型配列への参照 `primes` を，3 行目は現在格納されている素数の数を示す `int` 型の整数 `i` を，メンバ変数として定義している．これらのメンバ変数は，図 6.1 において永続ルートと指定された static フィールドからこのクラスのイン

```
1  import rbp.*;
2
3  public class Primes {
4    @durableroot
5    static PrimeGenerator generator;
6
7    public static void main(String[] args) {
8      // 永続オブジェクトのリカバリ or 初期化
9      Recovery.setNvmFile("/path/to/nvm");
10     if (Recovery.hasValidData()) {
11       ClassLoader[] clds = { /* 省略 */ };
12       Recovery.recovery(clds);
13     } else {
14       Recovery.init();
15     }
16
17     // アプリケーションの初期化
18     if (generator == null) generator = new PrimeGenerator();
19
20     // アプリケーションの実行
21     generator.generate();
22   }
23 }
```

図 6.1 Primes クラス

スタンスが参照されているので，到達可能性により永続化される．

5 行目から 9 行目は，メンバ変数の初期化を行うコンストラクタである．

12 行目の `isPrime` メソッドは，引数に与えた数が素数であるかどうかを返す．このメソッドの詳細は省略する．

14 行目から 26 行目は，素数の生成を行うアプリケーションの主部である．RBP は，このメソッドで利用するメンバ変数 `primes` と `i` への書き込みをプログラムの順序で自動的に永続化することを保証している．また，リカバリ処理によってメンバ変数がクラッシュ直前の内容に戻されて，処理を途中から再開できるようになる．ただし，そのデータを利用したアプリケーションの整合性は，Java プログラマが保証しなければならない．17 行目の `int` 型変数 `n` は，現在素数であるか調べている数である．変数の初期化は，途中から再開される

```
1  class PrimeGenerator {
2    int[] primes; // 素数を格納する配列
3    int i; // 現在格納した素数の数
4
5    // フィールドの初期化
6    PrimeGenerator() {
7      primes = new int[10000000];
8      i = 0;
9    }
10
11   // 素数判定
12   boolean isPrime(int n) { /* 省略 */ }
13
14   // 素数生成
15   void generate() {
16     // 現在素数であるか調べている数
17     int n = i == 0 ? 2 : primes[i - 1] + 1;
18     while (i < primes.length) {
19       if (isPrime(n)) {
20         primes[i] = n;
21         i++;
22       }
23       n++;
24     }
25   }
26 }
```

図 6.2 PrimeGenerator クラス

ことを考慮して適切に行わなければならない．ここでは，まだ素数の生成を開始していない場合（i $= 0$）は 2，既に開始している場合（i $\neq 0$）は生成されている素数の最大値に 1 を加算した値で初期化を行い，次に大きい素数を探すようにする．18 行目の while 文で，配列の要素が全て埋まるまで素数の生成を繰り返す．整数 n が素数である場合，20 行目と 21 行目の処理が行われる．永続化されるメンバ変数への書き込みには，整合性を保つため順序に注意する．ここでは，先に primes 配列にデータを格納した後に i をインクリメントすることで，primes 配列には昇順に i 個の素数が格納されていることを保証する．

第 7 章

性能評価

7.1 実験環境

7.1.1 計算機と VM

実験環境は以下の通りである．

- CPU: Intel Xeon Gold 6354 3.00GHz
- DRAM: DDR4 3200MHz 96GB
- NVM: Intel Optane DC Persistent Memory 200 Series 512GB
- OS: Ubuntu 20.04.4 LTS
- C Compiler: gcc version 9.4.0
- NOVA FileSystem DAX

CPU は 2 つのソケットに 18 コアずつ搭載されており，合計 36 コアである．この NUMA 環境におけるメモリアクセスで，インターコネクトを経由することによるレイテンシの低下を避けるため，実験では 1 つの CPU ソケットのみを使用し，これと同じ NUMA ノードに接続された NVM を使用する．

Java VM には OpenJDK 16 を使用し，“server” 構成でビルドする．JIT コンパイラは無効化しており，Template Interpreter のみで動作する．オブジェクトポインタとクラスポインタの圧縮機能は無効化している．GC は G1GC を選択する．

7.1.2 ベンチマークプログラム

永続化機構の性能を評価するベンチマークプログラムには，DaCapo Benchmark Suite [19] のバージョン dacapo-9.12-MR1-bach を使用する．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たベンチマークスイートである．ただし，batik，eclipse，tomcat，tradebeans，tradesoap は，RBP を実装していない元の OpenJDK 16 でも実行できないため使用しない．ベンチマークのサイズ（処理数）設定は default サイズを使用し，ウォームアップオプションを有効にする．各ベンチマークは 5 回ずつ計測し，各々の中央値を結果に示す．

表 7.1 は，各ベンチマークの名前と測定した統計データである．‘実行時間’ の列は，JIT コンパイラを無効化した RBP を実装していない元の VM での実行時間である．‘書き込み頻度’ の列は，1 秒間に行われた書き込みの回数を示す．これは，通常のフィールド（‘ordinary’ の列）と `volatile` フィールド（‘volatile’ の列）で個別にカウントした．さらに，‘アトミック操作’ の列には，`volatile` フィールドの書き込みのうち，アトミック操作を行った回数を示す．

リカバリ機構の性能評価には，6.1 節で示した文字列生成アプリケーションを用いる．詳細は 7.5 節で後述する．

7.1.3 永続ルートの選択

永続化機構の評価に利用する DaCapo Benchmark Suite では，我々が static フィールドから永続ルートを選択すると，選択に偏りが発生する危険性がある．そこで，永続ルートの選び方は，永続ルートを全く選択しない設定（none-durable）か，全ての static フィールドを永続ルートとして選択する設定（all-durable）の二択とする．

リカバリの評価に利用する文字列生成アプリケーションでは，リカバリ後に文字列の生成を再開できる程度の最低限のデータを保持できるように，永続ルートを選択する．また，リカバリの評価は永続オブジェクトのサイズ毎のリカバリ時間を測定して比較するため，永続

表 7.1　ベンチマークと統計データ

ベンチマーク	実行時間 (秒)	書き込み頻度 (10^6 回/秒)		アトミック操作 (10^6 回/秒)
		ordinary	volatile	
avrora	11.7	37.40	0.12	0.00
fop	5.1	5.16	0.01	0.00
h2	70.2	2.05	0.01	0.01
jython	56.5	5.18	0.15	0.04
luindex	8.6	11.28	0.00	0.00
lusearch	2.1	174.38	0.01	0.00
lusearch-fix	2.1	174.38	0.01	0.00
pmd	5.0	22.51	0.02	0.00
sunflow	9.2	86.66	0.00	0.00
xalan	2.7	95.70	0.03	0.00

ルートの選択の偏りは問題にならない.

7.2　基本的なオーバヘッド

RBP は，永続オブジェクトを作成しないプログラムでも，**基本的な**オーバヘッドを発生させる．例えば，ライトバリアは，オブジェクトが永続オブジェクトであるかどうかを確認する必要がある．これは，`replica` フィールドから値を読み出すことで行い，図 4.8 に示すように `MFENCE` 命令を含む．また，RBP は dram copy のオブジェクトヘッダに 1 ワードの `replica` フィールドを追加するため，GC の頻度が増加する可能性がある．さらに，`volatile` フィールドへの書き込みとアトミック操作では，4.4.7 節で説明したスピンロックが必要となる.

基本的なオーバヘッドを調査するために，none-durable の設定を用いて，オブジェクトを

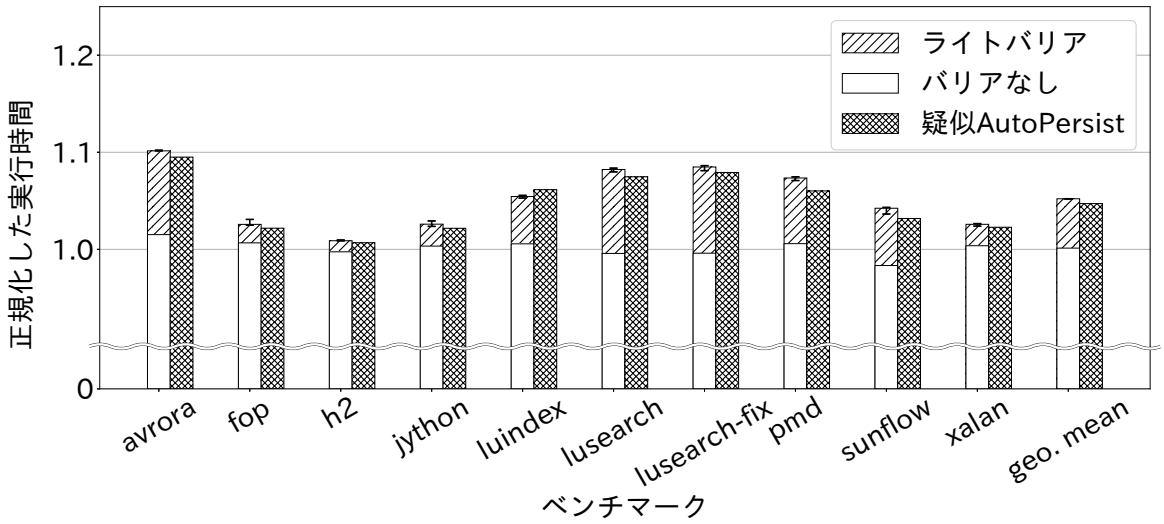


図 7.1 none-durable の設定における実行時間と内訳

全く永続化しない場合の実行時間を計測した．図 7.1 の左側の棒グラフは，RBP を実装していない元の VM を 1 に正規化した実行時間と，その内訳を示している．‘ライトバリア’ の部分は，ライトバリアによるオーバヘッドを示している．つまり，永続オブジェクトであるかどうかの確認と，`volatile` フィールドやアトミック操作のスピンロックのオーバヘッドである．ライトバリアによるオーバヘッドは，平均で 5.2%であった．

このライトバリアによるオーバヘッドのほとんどは，永続オブジェクトであるかどうかの確認によるものであった．表 7.1 に示すように `volatile` フィールドやアトミック操作は少ないため，スピンロックのオーバヘッドは小さく，最大で 0.9%，平均で 0.3%であった．

その他の部分の実行時間は平均で 100.1%であり，dram copy に 1 ワードの `replica` フィールドを導入することによるオーバヘッドは無視できるほど小さい．

7.3 AutoPersist との比較

先行研究の AutoPersist のライトバリアの一部を OpenJDK 16 に実装し，RBP とオーバヘッドを比較した．この部分実装を**疑似** *AutoPersist* と呼ぶ [*1]．疑似 AutoPersist には，

[*1] 論文の説明から完全な AutoPersist を実装するのは困難であるため，AutoPersist のライトバリアの一部を実装した．

AutoPersist のリードバリアを実装していない点に注意されたい.

AutoPersist [6] は，永続化を伴わない場合の追加処理のオーバヘッドについて，“このオーバヘッドは平均して実行時間の 10%未満であることが，QuickCheck [20] によって示されている” と述べている．しかし，AutoPersist の評価環境は，本研究とは大きく異なっている．まず，AutoPersist は実験用の Java VM である Maxine VM [21] で実装されていたのに対し，我々は OpenJDK で RBP を実装した．また，AutoPersist は JIT コンパイラを用いて実行されていた．さらに，マルチスレッド環境におけるスレッドセーフの仕組みは，QuickCheck では説明されていなかった.

AutoPersist のライトバリアは，まずオブジェクトが移動されているかどうかを確認する．移動されていない場合，値を dram copy に書き込む．AutoPersist の論文 [6] の 6.3 節によると，書き込み時にオブジェクトが移動されたかどうかを確認する．この確認は，専用のオブジェクトヘッダを読み込むことによって行われる．値の書き込みとヘッダの読み込みの間には，メモリバリアが挿入される．このメモリバリアは，かなりのオーバヘッドとなる可能性がある.

疑似 AutoPersist は，メモリバリアのオーバヘッドを再現する形で実装した．疑似 AutoPersist の実装の詳細は次の通りである．各オブジェクトヘッダに，空の 1 ワードのフィールドを追加した．このヘッダは，オブジェクトが作成されたときにゼロで初期化され，以後変更されない．ライトバリアでは，まずオブジェクトに値を書き込む．次に，`MFENCE` 命令を発行し，空のヘッダから読み出し，ヘッダがゼロでない場合に追加の処理を行う条件分岐命令を発行した．勿論，この条件は満たされない．しかし，ヘッダの読み出しが完了するまで条件分岐の分岐先が決定できないため，投機的に実行される命令がある．疑似 AutoPersist には，ライトバリアでアクセスされたオブジェクトの現在の位置を取得するプロセスなど，AutoPersist の他の機構は実装しない.

図 7.1 の右側の棒グラフは，疑似 AutoPersist の実行時間を示している．疑似 AutoPersist のオーバヘッドは平均して 4.7%であった．RBP (左側の棒グラフ) と比較すると，疑似 AutoPersist のライトバリアのオーバヘッドは RBP よりもわずかに小さく，平均して 0.5%

ほど小さかった．疑似 AutoPersist では AutoPersist のライトバリアの一部のみを実装し，リードバリアは実装していない．したがって，疑似 AutoPersist は AutoPersist のオーバヘッドの下限を示している．AutoPersist の実際のオーバヘッドは，オブジェクトからの読み出しに対してリードバリアを用いた追加処理が必要であるため，疑似 AutoPersist よりも大きくなると考えられる．したがって，RBP の基本的なオーバヘッドは，AutoPersist と同程度かそれ以下であることが期待される．

7.4　永続化コスト

all-durable の設定を用いて，オブジェクトの複製の作成と，nvm copy の内容を dram copy と同じものに保つための nvm copy への書き込みにかかるコストを計測した．その結果を，表 7.2 に示す．‘複製の作成’ の列は，1 秒間あたりの複製を作成した回数（‘10^3 個/秒’ の列）とサイズ（‘10^3 ワード/秒’ の列）を示している．‘永続オブジェクトへの書き込み’ の列は，1 秒あたりの永続オブジェクトへの書き込み回数を示している．このとき，nvm copy への書き込みが発生する．‘生存している永続オブジェクトの割合’ の列は，生存しているオブジェクトの合計サイズに対する，生存している永続オブジェクトの合計サイズの割合である．生存しているとは，GC によって回収されずに残っていることを指し，GC が終わったタイミングで計測する．計測はバイトサイズ単位で行い，1 回のベンチマーク実行における平均値を示している．全てのオブジェクトは dram copy を持っており，これに加えて永続オブジェクトは nvm copy を持っている．したがって，永続オブジェクトの割合はメモリのオーバヘッドを表している．

図 7.2 は，all-durable の設定における実行時間と，その内訳を示している．縦軸は，RBP を実装していない元の VM を 1 に正規化した実行時間である．グラフの ‘実行’ の部分は，7.2 節で説明した基本的なオーバヘッドが含まれている．グラフの ‘複製の作成’ の部分は，図 4.7 で説明した `make_persistent` の実行時間を示している．グラフの ‘NVM への書き込み’ の部分は，永続オブジェクトの nvm copy への書き込みにかかった時間を示している．

表 7.2　複製の作成頻度と nvm copy への書き込み頻度

ベンチマーク	複製の作成		永続オブジェクトへの書き込み (10^6 回/秒)	生存している永続オブジェクトの割合 (%)
	(10^3 個/秒)	(10^3 ワード/秒)		
avrora	10.5	65.4	15.6	83.5
fop	20.6	245.3	0.4	26.9
h2	35.8	362.9	0.9	98.4
jython	213.0	2,176.0	1.0	92.6
luindex	4.2	140.1	3.8	88.6
lusearch	186.5	1,022.6	25.7	70.5
lusearch-fix	186.7	1,023.6	25.7	70.5
pmd	165.3	1,717.5	0.1	58.0
sunflow	2.2	31.0	0.0	28.7
xalan	257.0	13,271.1	26.5	89.5

ここには，CLWB 命令と SFENCE 命令によるキャッシュラインの書き戻しにかかる時間も含まれている．

実験結果より，複製を作成してオブジェクトを永続化するコストは，比較的小さいことがわかる．表 7.2 で示すように，オブジェクトの複製を作成する頻度は，個数が 257.0×10^3 個/秒，サイズが $13,271.1 \times 10^3$ ワード/秒である xalan が最大であった．その xalan においても，make_persistent のオーバヘッドは 6.9%に留まった．

一方，永続オブジェクトへの書き込みには大きなコストを要した．特に，ベンチマークプログラムの avrora，luindex，lusearch，lusearch-fix，xalan は，1 秒あたり 1.0×10^6 回以上の頻繁な永続オブジェクトへの書き込みを行っており，オーバヘッドは 100%以上を超えた．他のベンチマークプログラムでは永続オブジェクトへの書き込みが少ないため，nvm copy への書き込みコストは小さかった．

永続オブジェクトからの読み出し処理は，非永続オブジェクトと同じくらい高速に行うこ

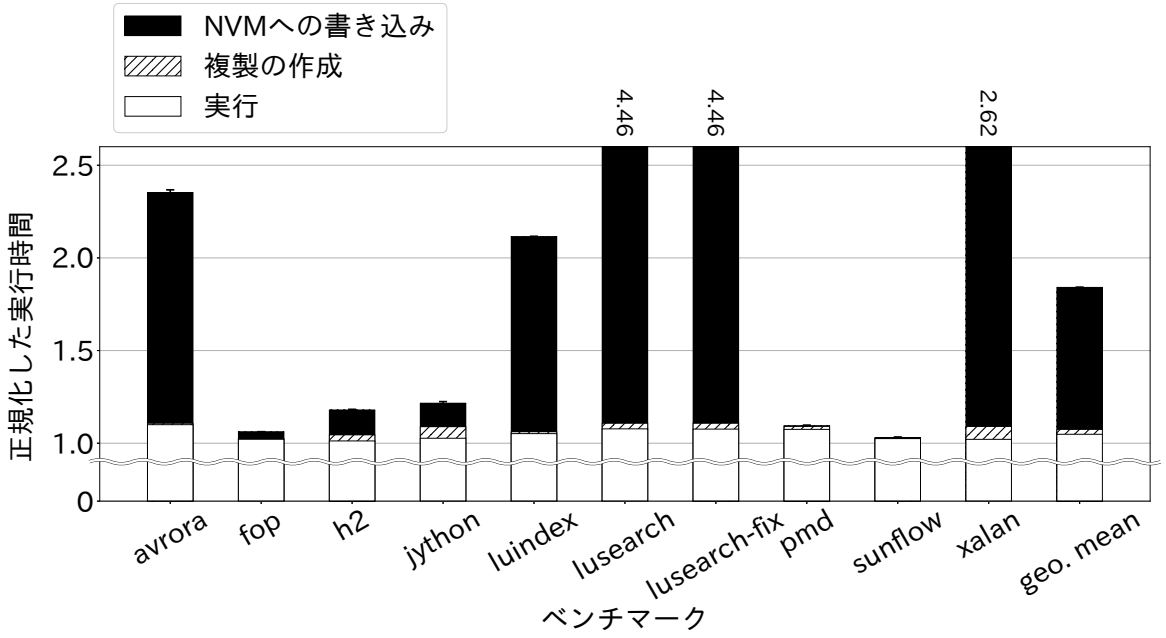


図 7.2　all-durable の設定における実行時間と内訳

とができた．これは，RBP が，永続オブジェクトであるかどうかに関わらず，リードバリアを使わずに dram copy から読み出すためである．実際，図 7.2 の ‘実行’ の部分の時間と，図 7.1 の左側の棒の時間は同程度であり，永続オブジェクトからの読み出しによるオーバヘッドは観測されていない．

7.5　リカバリ時間

6.1 節で示した文字列生成アプリケーションを用いて，永続オブジェクトのサイズによるリカバリ時間の違いを比較して，リカバリの性能を調査した．永続化する文字列の数を 100 万個から 1000 万個まで 100 万個ずつ増加させた 10 通りのバリエーションを用意した．計測は 5 回ずつ行い，各々の中央値を結果として示す．

図 7.3 に結果を示す．横軸は永続化した文字列の個数（$\times 10^6$ 個），縦軸はリカバリ時間（秒）である．

実験結果より，永続化する文字列の個数に比例してリカバリ時間が増加していることがわかる．要素数が 100 万個（約 472MB）増加すると，リカバリ時間は平均で 1.97 秒増加す

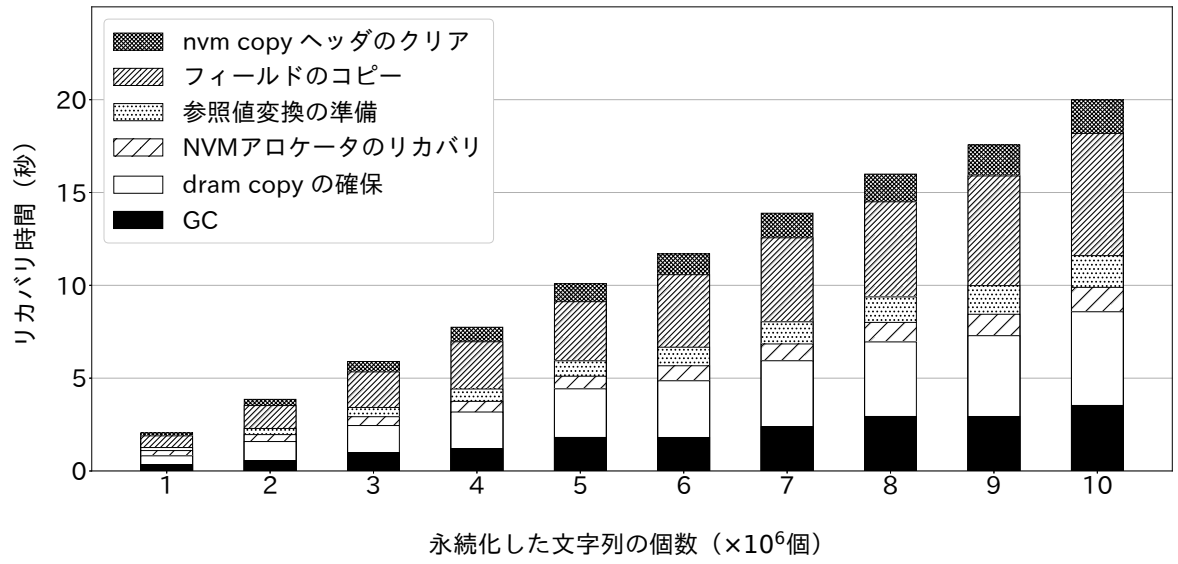


図 7.3　リカバリ時間と内訳

る．1GB あたりに換算すると 4.18 秒増加することになる．黒色のグラフは，dram copy を確保する際に発生した GC の実行時間を示している．GC 時間を除くと，要素数 100 万個あたりのリカバリ時間の増加量は平均 1.64 秒であり，1GB あたりに換算すると 3.46 秒になる．今回調査した最大サイズである 1000 万個の文字列（約 4.72GB）のリカバリ時間は 20.0 秒であり，GC 時間を除くと 16.5 秒であった．

積み上げグラフは，リカバリ時間の内訳を示している．このグラフの凡例の下から上に向かって順にリカバリ処理が進む．5.3.1 節で述べたクラスのロードは，要素数に関わらず 67 ミリ秒でほぼ一定であり，実行時間も短いため実験結果のグラフには含まれていない．dram copy の確保は 5.3.2 節，NVM アロケータのリカバリは 5.3.3 節，参照値変換の準備は 5.3.4 節，フィールドのコピーと nvm copy ヘッダのクリアは 5.3.5 節で述べた処理である．実験結果より，dram copy の確保とフィールドのコピー処理に時間がかかっていることがわかる．また，dram copy の確保にかかる時間のうち 4 割程度が GC によるものであることがわかる．

第 8 章

関連研究

プログラミング言語で NVM を利用する研究には，主に二つの方向性がある．一つは，NVM を DRAM の拡張として使用することである [22, 23]．それらの研究の主な目的は永続化ではなく，アプリケーションの総スループットを向上させること，システムの総消費電力を減らすこと，NVM デバイスの寿命を延ばすことである．もう一つの方向性は，データを永続化することである．

Mnemosyne [24] や NV-Heaps [25] は，NVM をトランザクションに統合した初期のシステムである．その後，永続的なトランザクションの性能を向上させる研究が行われた [26, 27, 28, 29, 30]．Atlas [31] は NVM への永続化をロックに統合した．このプログラミングモデルについては，別の論文 [32] で議論されている．マネージドランタイムは，より強力な統合を可能にする．例えば，通常の書き込み（変数への代入）操作によって書き込まれたデータを，自動的にフラッシュすることができる．Java VM でこれを実現した研究には，Espresso [33] や AutoPersist [6] がある．Espresso [33] は，マネージドランタイムシステムである Java VM に NVM を統合したが，トランザクションを用いる．Shull らは，到達可能性に基づく永続化のプログラミングモデル [17] を提案し，その Java VM 実装を AutoPersist [6] として示した．AutoPersist は，実験用の Java VM である Maxine VM [21] に実装されている．RBP は，製品レベルの Java VM である OpenJDK 16 に実装されている．

NV-Heaps [25] や NV-HTM [27] は，プログラムに遅延を挿入することで NVM をエミュレートして性能を評価しており，実機は用いられていない．Espresso [33] は，実機の 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．AutoPersist [6] は NVM の実機を用いて Espresso との比較を行っているが，オブジェクトを永続化しない元の VM との比較は示されていない．本研究では，NVM の実機を用いて RBP のオーバヘッドを計測し，RBP を実装していない元の VM や，AutoPersist のライトバリアの一部を部分的に実装した疑似 AutoPersist と比較する形で実験結果を示した．

ログベースのトランザクションを用いるアプローチ [25, 27, 30] は，AutoPersist や RBP に比べて 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．AutoPersist や RBP 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ことを重視したアルゴリズムであり，キャッシュラインを書き込み毎に書き戻す．また，データの整合性は Java プログラマが保証しなければならない．その一方で，トランザクションや永続データ構造などを，Java で直感的かつ自由に実装可能であるというメリットを持つ．

AutoPersist では，リードバリアを用いたオブジェクトの移動によるアプローチを取っているが，我々はリードバリアを用いないオブジェクトの複製によるアプローチを取っている．AutoPersist のバリアのコストは，QuickCheck [20] の論文で議論された．これでは，JIT コンパイラを用いた Maxine VM で平均して 8.8%のバリアに関するオーバヘッドがあると主張された．しかし，QuickCheck はメモリバリア命令のコストを扱っていない．つまり，リードバリアとライトバリアでオブジェクトが転送されたかどうかをチェックする処理と，ライトバリアにおけるメモリバリア命令を除く処理のみを扱っている．P-INSPECT [34] は，バリアに関するオーバヘッドを緩和するため，プロセッサ自体の拡張を提案した．GCPersist [35] は，オブジェクトを直ちに永続化することを諦め，永続化のコストを削減した．

複製のアプローチは，C と C++ のライブラリとしても研究されている．NV-HTM [27], DudeTM [36]，Kamino-TX [37] は，failure atomic トランザクションを提供する．これら

は，NVM 上に DRAM 領域のページなどの複製を作成する．PMThreads [38] も，DRAM のページの複製を作成する．これは，ロックベースの並列アルゴリズム用に設計されている．Mirror [7] は，ロックフリーのデータ構造の複製を作成し，DRAM と NVM のコピーをロックフリーな方法で保持する．我々は，マネージドランタイム用の複製を用いたアルゴリズムを提案した．

到達可能性に基づく永続化モデルは，NVM が登場する前に Orthogonal persistency の概念 [4, 39, 40] の一部として提案された．Orthogonal persistency [4] は，汎用的な永続ストレージを用いてオブジェクトを永続化するシステムを提案した．また，その著者は PJama プラットフォーム [5] を実装した．

本研究では，永続データへのアクセスがデータレースフリーであることを仮定しているが，これは他のシステムでも同様に必要となる．Atlas [31] は，明示的にそれを記述している．AutoPersist [6] も，厳格な永続性（strict persistency） [41] を満たすために必要である．

RBP では，いくつかの GC に関する技術を利用している．複製は，Nettles らが提案した GC [42] で導入された．dram copy と nvm copy の内容が一致するよう両方に書き込みを行う**ダブルアップデートバリア**は，複製に基づく GC [18, 43, 44] で使用されている．本研究では，Transactional Sapphire [18] の並行コピープロトコルを使用した．これは，通常の書き込み処理において，NVM に書き込む以外のオーバヘッドを課すことはない．並行コピーの別のアプローチとして，複製を作成する処理がオブジェクトをコピーする前にオブジェクトヘッダのフラグをセットし，書き込み処理はオブジェクトヘッダのフラグをクリアすることで，複製を作成する処理がコピー後に競合を検出するものがある．このアプローチは，書き込み操作で最新の状態を読む必要があるが，弱いメモリ一貫性モデルではメモリバリアが必要であるため，追加のコストがかかる．AutoPersist は，ライトバリアでメモリバリア命令を用いることにより，このアプローチを採用した．並行コピー GC アルゴリズムである Chicken [45] と Staccato [46] は，GC の Safepoint を用いて，書き込み操作で最新の状態が非常に小さなコストで読み込まれることを保証している．ただし，コピーするスレッ

ドは，他のすべてのスレッドが GC の Safepoint に到達するのを待たなければならない．この手法を RBP で利用することは不可能である．

本研究の環境では，NVM がファイルシステムに抽象化されており，mmap システムコールを用いて仮想メモリ空間にマップする．アドレスの衝突などが原因で前回の実行とは異なるアドレスにマップされた場合は，フィールドに保持している参照値を修正する必要がある．これに対処した研究 [33, 47] は，前回のマッピングアドレスを永続化しておき，今回のマッピングアドレスとの差分を使って参照値を修正する．この修正中にクラッシュが発生するとデータの整合性が失われるため，undo ログを作成する必要がある．本研究はこの参照値を修正する機構を備えておらず，実装は今後の課題である．ただし，64 ビット OS の大きなアドレス空間ではアドレス衝突は稀であると考えられる [33, 47].

実行速度の改善を図るために，ヒープの走査等で復元できるアロケータ情報を永続化しないシステム [7, 30, 47] がある一方，Espresso [33] のようにアロケータ情報を永続化することでリカバリ時にヒープを走査しないシステムも存在しており，これらはトレードオフの関係にある．本研究のリカバリ処理ではヒープの走査は避けられないため，アロケータ情報を極力永続化しない前者のアプローチを取っている．

リカバリ処理はアプリケーションの実行開始を遅らせるため，それを改善する手法が提案されている．複製に基づく永続化を行う PMThreads [38] は，OS の copy-on-write 機構を用いて NVM から DRAM へのデータコピー処理を遅延させることで高速化を図る．具体的には，mmap システムコールの MAP_PRIVATE フラグを用いて NVM ファイルをマッピングし，それを DRAM データとして扱う．この機構は，書き込みが発生するまで NVM から読み出しを行う．書き込みが発生すると NVM データの複製を DRAM 上に作成し，その後の読み書きには DRAM を利用する．

リカバリ処理とアプリケーションの実行を並行して行う手法 [47] も提案されている．mprotect システムコールを用いて，リカバリスレッド以外のスレッドがリカバリ中の NVM 領域にアクセスするのを禁止する．アプリケーションスレッドがリカバリ中の NVM 領域にアクセスした場合，シグナルが発生する．シグナルを捕捉したアプリケーションス

レッドは，リカバリスレッドの代わりに，そのアクセスしたNVM領域のリカバリ処理を実行する．

本研究では，リカバリを遅延させたり，リカバリ中に永続オブジェクトにアクセスする並行動作の機能を有していない．ただし，永続オブジェクトにアクセスしない処理は，5.3.4節と5.3.5節で述べた「参照値変換の準備」と「フィールドデータのコピー」を除いて並行動作させることができる．7.5節の実験結果より，リカバリ時間の31.0%は他のスレッドと並行動作が可能である．他のスレッドでは，永続オブジェクトに関係ない初期化処理などを実行することが期待される．

第 9 章

おわりに

本研究では，到達可能性による永続化を実現するための新しいアルゴリズムである RBP を提案した．マルチスレッド環境で発生する可能性がある競合状態に焦点を当て，そのアルゴリズムを説明した．先行研究の AutoPersist とは異なり，RBP は永続オブジェクトからの読み出しにリードバリアや NVM へのアクセスが不要である．オブジェクトを永続化して NVM にデータを残す永続化機構に加えて，クラッシュ後に NVM 上のデータを元に DRAM 上にオブジェクトを再構築するリカバリ機構も提案した．評価のために，RBP を製品レベルの Java VM である OpenJDK 16 に実装した．永続オブジェクトからの読み出しは，非永続オブジェクトからの読み出しと同じくらい高速であることを確認した．オブジェクトの複製を作成するためのコストは小さく，永続オブジェクトへの書き込みは遅く，永続化を全く行わないプログラムでも発生するライトバリア等の基本的なオーバヘッドは，AutoPersist と同程度であることがわかった．リカバリ処理には，永続化したオブジェクトのサイズ 1GB あたり約 4.72 秒の時間が必要であった．さらに，実際に提案システム上で動作する Java アプリケーションをクラッシュさせ，正しく復元できることを確認した．

謝辞

本研究を行うにあたり，丁寧なご指導をいただいた高知工科大学の高田喜朗准教授，東京大学の鵜川始陽准教授と明治大学の岩崎英哉教授に深く感謝致します．また，副査をしてくださった高知工科大学の横山和俊教授と松崎公紀教授に心より御礼申し上げます．共同研究者である電気通信大学大学院の長安尚之さんをはじめ，研究室メンバと支えてくださった全ての人に感謝致します．

参考文献

[1] 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/products/docs/memory-storage/optane-persistent-memory/overview.html. 2023-01-10 参照.

[2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The Java® Language Specification Java SE 16 Edition. https://docs.oracle.com/javase/specs/jls/se16/html/index.html, 2021.

[3] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java® Virtual Machine Specification Java SE 16 Edition. https://docs.oracle.com/javase/specs/jvms/se16/html/index.html, 2021.

[4] Malcolm P. Atkinson and Ronald Morrison. Orthogonally Persistent Object Systems. *VLDB J.*, 4(3):319–401, 1995.

[5] Malcolm P. Atkinson and Mick J. Jordan. Issues Raised by Three Years of Developing PJama: An Orthogonally Persistent Platform for Java. In *Proceedings of the 7th International Conference on Database Theory (ICDT 1999)*, volume 1540 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 1999.

[6] Thomas Shull, Jian Huang, and Josep Torrellas. AutoPersist: an easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*, pages 316–332. ACM, 2019.

[7] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*, pages 1218–1232. ACM, 2021.

[8] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 378–391. ACM, 2005.

[9] 長安 尚之, 鵜川 始陽, 松本 康太郎, 岩崎 英哉. 複製に基づく永続化を行うシステムにおける不揮発性メモリ管理手法. 第 24 回プログラミングおよびプログラミング言語ワークショップ (PPL 2022), 2022.

[10] Intel. Intel ® Optane ™ DC Persistent Memory Start Up Guide Revision 2.0. `https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf`, 2020.

[11] Jian Xu and Steven Swanson. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. USENIX Association, 2016.

[12] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Andy Rudoff, and Steven Swanson. Hardening the NOVA File System. Technical Report CS2017-1018, Department of Computer Science & Engineering, University of California, San Diego, May 2017.

[13] The Non-Volatile Systems Lab. NOVA: NOn-Volatile memory Accelerated log-structured file system. `https://github.com/NVSL/linux-nova`. 2023-01-10 参照.

[14] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2 (2A, 2B, 2C, & 2D): Instruction Set Reference, A–Z*, 2021.

[15] Ian Piumarta and Fabio Riccardi. Optimizing Direct Threaded Code by Selective Inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 291–300, 1998.

[16] Edsger W. Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[17] Thomas Shull, Jian Huang, and Josep Torrellas. Defining a high-level programming model for emerging NVRAM technologies. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang 2018)*, pages 11:1–11:7. ACM, 2018.

[18] Tomoharu Ugawa, Carl G. Ritson, and Richard E. Jones. Transactional Sapphire: Lessons in High-Performance, On-the-fly Garbage Collection. *ACM Trans. Program. Lang. Syst.*, 40(4):15:1–15:56, 2018.

[19] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2006)*, pages 169–190. ACM, 2006.

[20] Thomas Shull, Jian Huang, and Josep Torrellas. QuickCheck: using speculation to reduce the overhead of checks in NVM frameworks. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2019)*, pages 137–151. ACM, 2019.

[21] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2017)*, pages 74–82. ACM, 2017.

[22] Chenxi Wang, Ting Cao, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. Efficient Management for Hybrid Memory in Managed Language Runtime. In *Proceedings of the 13th IFIP WG 10.3 International Conference on Network*

and Parallel Computing (NPC 2016), volume 9966 of *Lecture Notes in Computer Science*, pages 29–42, 2016.

[23] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 62–77. ACM, 2018.

[24] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 91–104. ACM, 2011.

[25] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 105–118. ACM, 2011.

[26] Hillel Avni, Eliezer Levy, and Avi Mendelson. Hardware Transactions in Nonvolatile Memory. In *Proceedings of the 29th International Symposium on Distributed Computing (DISC 2015)*, volume 9363 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2015.

[27] Daniel Castro, Paolo Romano, and João Pedro Barreto. Hardware Transactional Memory Meets Memory Persistency. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium, (IPDPS 2018)*, pages 368–377. IEEE Computer Society, 2018.

[28] Alexandro Baldassin, Rafael Murari, João P. L. de Carvalho, Guido Araujo, Daniel Castro, João Barreto, and Paolo Romano. NV-PhTM: An Efficient Phase-Based Transactional System for Non-volatile Memory. In *Proceedings of the 26th European*

Conference on Parallel and Distributed Computing (Euro-Par 2020), volume 12247 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2020.

[29] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: efficient, HTM-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2020)*, pages 59–74. ACM, 2020.

[30] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. Association for Computing Machinery, 2020.

[31] Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*, pages 433–452. ACM, 2014.

[32] Hans-Juergen Boehm and Dhruva R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*, pages 55–67. ACM, 2016.

[33] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018)*, pages 70–83. ACM, 2018.

[34] Apostolos Kokolis, Thomas Shull, Jian Huang, and Josep Torrellas. P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2020)*, pages 509–524. IEEE, 2020.

[35] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. GCPersist: an efficient GC-assisted lazy persistency framework for resilient Java applications on NVM. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2020)*, pages 1–14. ACM, 2020.

[36] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017)*, pages 329–343. ACM, 2017.

[37] Amir Saman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys 2017)*, pages 499–512. ACM, 2017.

[38] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, pages 623–637. ACM, 2020.

[39] Malcolm P. Atkinson, Mick J. Jordan, Laurent Daynès, and Susan Spence. Design Issues for Persistent Java: A Type-Safe, Object-Oriented, Orthogonally Persistent System. In *Proceedings of the 7th Workshop on Persistent Object Systems*, pages 33–47. Morgan Kaufmann, 1996.

[40] Malcolm P. Atkinson, Laurent Daynès, Mick J. Jordan, Tony Printezis, and Susan Spence. An Orthogonally Persistent Java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[41] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA 2014)*,

pages 265–276. IEEE Computer Society, 2014.

[42] Scott Nettles and James W. O'Toole Jr. Real-Time Replication Garbage Collection. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI 1993)*, pages 217–226. ACM, 1993.

[43] Richard L. Hudson and J. Eliot B. Moss. Sapphire: copying garbage collection without stopping the world. *Concurr. Comput. Pract. Exp.*, 15(3-5):223–261, 2003.

[44] Tomoharu Ugawa, Hideya Iwasaki, and Taiichi Yuasa. Improved replication-based incremental garbage collection for embedded systems. In *Proceedings of the 9th International Symposium on Memory Management (ISMM 2010)*, pages 73–82. ACM, 2010.

[45] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A Study of Concurrent Real-Time Garbage Collectors. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 33–44. ACM, 2008.

[46] Bill McCloskey, David F. Bacon, Perry Cheng, and David Grove. Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors. IBM Research Report RC24504, IBM Research, 2008.

[47] Nachshon Cohen, David T. Aksun, and James R. Larus. Object-Oriented Recovery for Non-Volatile Memory. 2018.