不揮発性メモリを用いた 複製に基づく永続化のためのリカバリ機構

松本 康太郎[1,a)]　長安 尚之[2,b)]　鵜川 始陽[3,c)]　高田 喜朗[1,d)]　岩崎 英哉[4,e)]

概要：不揮発性メモリ（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．我々は，プログラム実行中に DRAM 上のオブジェクトを NVM 上に複製することで，システムがクラッシュした後でもデータを復元できる Java 仮想機械（VM）を開発している．これまで，オブジェクトの複製を作る機能を実現したが，本研究ではさらに，クラッシュ後のプログラムの再起動において，複製を使ってデータを復元するリカバリ機能を実現した．そのために，クラスオブジェクトなどのメタ情報を NVM 上に保存する仕組みを開発した．リカバリ処理では，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．この処理を，ランタイムシステムに C++で実装した関数と Java のライブラリメソッドが協調して行う．これにより，リカバリ処理中にごみ集めが起きるような複雑なケースにも対応している．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．

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，リカバリ，Java VM

1. はじめに

不揮発性メモリ（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．これを「永続化」されている，という．

[1] 高知工科大学 大学院工学研究科
[2] 電気通信大学 大学院情報理工学研究科
[3] 東京大学 大学院情報理工学系研究科
[4] 明治大学 理工学部
[a)] 255119t@gs.kochi-tech.ac.jp
[b)] 20nnagayasu@ipl.cs.uec.ac.jp
[c)] tugawa@acm.org
[d)] takata.yoshiaki@kochi-tech.ac.jp
[e)] hideya.iwasaki@acm.org

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．また，DRAM のデータは揮発性であるため，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．

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．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．

Orthogonal persistency [1] 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，システム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．我々は，オブジェクトの到達可能性に基づいて Java VM でこれを実現する Replication Based Persistency（以後，RBP と呼ぶ）の研究 [6] を行っている．到達可能性に基づく永続化とは，特定の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全てのオブジェクトを永続化対象とする考え方である．RBP は，Java プログラマがアノテーションを用いて指定した static フィールド（これを永続ルートと呼ぶ）を開始点とし，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．

この永続化機構により，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．この問題を解決するためには，NVM 上のデータを元に DRAM 上のデータを完全に復元して，クラッシュ前の状態に戻す必要がある．本研究では，これを実現するリカバリ機構を提案する．リカバリ機構は Java API として提供する．Java プログラマはこれを，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データの内容を 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ごみ集め（ガベージコレクション，GC）に対処しなければならない．そのため，一部の処理を Java ライブラリメソッドとして実装し，ランタイムシステムに C++で実装した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．

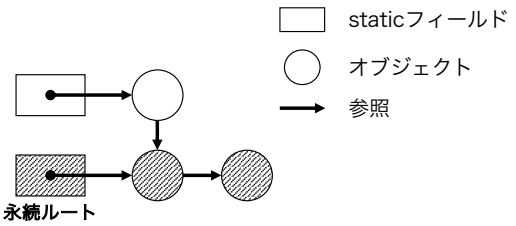

図 1 到達可能性に基づく永続化モデル

永続化機構が実装された OpenJDK 16 [5][7] にリカバリ機構を追加実装して，複数のプログラムで動作確認を行った．また，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を調査した．

2. 前提知識

2.1 到達可能性に基づく永続化

Orthogonal persistency [1] を構成する原則の一つに，「永続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 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．

2.2 永続化機構

RBP は，DRAM 上にあるオブジェクトの領域（dram copy）の複製を NVM 上の領域（nvm copy）

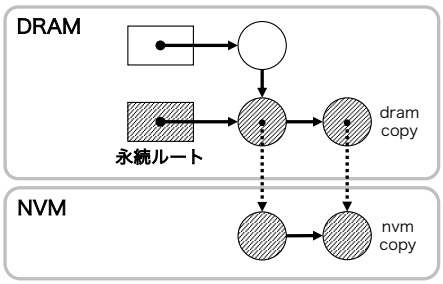

図 2 dram copy と nvm copy

に作成することで永続化を実現する．dram copy と nvm copy は，常に同じ最新の値を保持するようにする．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 へのポインタを持たせる．これを転送ポインタと呼ぶ．

図 2 に，dram copy の複製である nvm copy を NVM 上に作成した様子を示す．点線矢印は，dram copy から nvm copy への転送ポインタである．

2.2.1 永続オブジェクトへのアクセス

dram copy と nvm copy の両方に同じ最新の値を持たせるため，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．一方，読み出しは永続オブジェクトであるか否かに関わらず，常に dram copy から行う．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．

nvm copy への書き込みは dram copy のヘッダにある転送ポインタを介して行う．nvm copy に参照値を書き込む場合は，dram copy を指す参照から nvm copy を指す参照に変換する必要がある．変換を行わなければ nvm copy から dram copy を指す参照が発生し，クラッシュ後に dangling pointer となってしまうためである．また，書き込む参照値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 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．

2.2.2 複製の作成

複製の作成は，2.3.1 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．フィールドのコピーで 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．

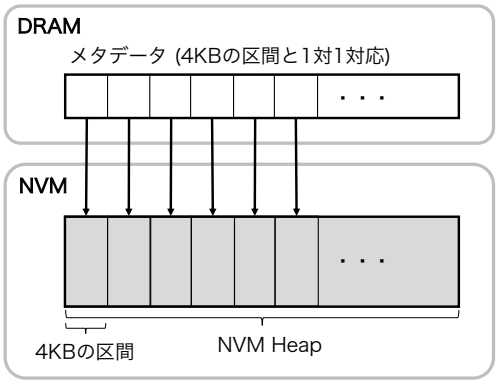

図 3 NVM 領域の管理機構

2.3 NVM 領域の管理

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．

2.3.1 NVM アロケータ

NVM アロケータはオブジェクトに，あらかじめ決めてある 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 3 は NVM アロケータの概略である．NVM 上のヒープ（3.1 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・解放を行う．メタデータはアロケーションの状態を記憶する領域や，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，NVM に置

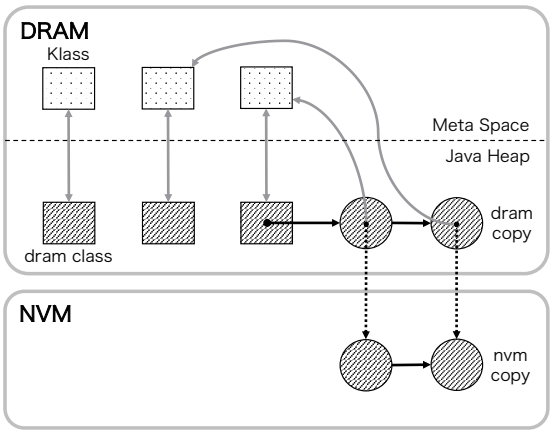

図 4 OpenJDK と既存の永続化機構が作成するデータ

かれた情報から復元することができるので，4.2.3 節で述べる復旧手順により整合性は保たれる．

Java スレッドは，各サイズクラスに対応した NVM 領域の区間，および 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．

特に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 領域上に配置する．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．

2.3.2 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しなければならず，オーバヘッドとなり得る．マーク&スイープではオブジェクトは移動しないため，GC に整合性が失われることはない．

2.4 static フィールドとクラス情報

図 4 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

[*1] OpenJDK では Mirror オブジェクトと呼ばれているが，nvm copy と名前がまぎ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 5 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

内部で Java クラス情報を保持するデータ Klass を示す．

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 VM 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 5 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．

3. 提案手法

リカバリ機構は Java API として提供する．本節では，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，提供するリカバリ API について記す．

3.1 永続化機構の改造

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

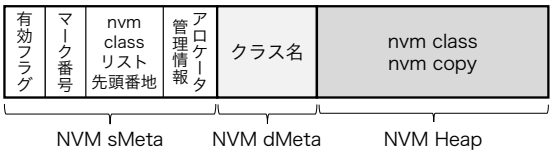

図 6 NVM 領域の内訳

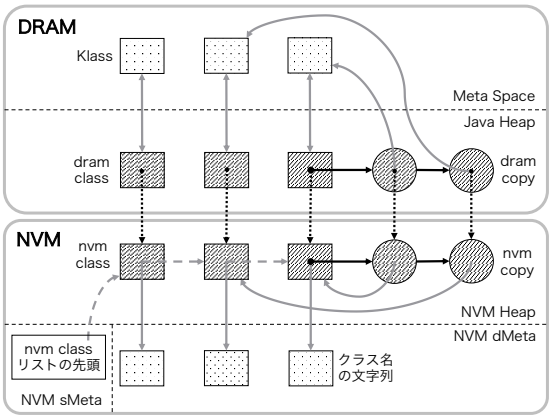

図 7 OpenJDK と改造した永続化機構が作成するデータ

- 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 6 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，2.3.1 節で示した NVM アロケータによって管理する．

3.1.1 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，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 7 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．

3.1.2 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 7 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．

3.2 リカバリ API

リカバリ機構は Java API として提供する．主な API は，以下の 4 個である．

- `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である．

リカバリ API の使用例を図 8 に示す．1 行目で

```
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 8 リカバリ API の使用例

インポートするパッケージには，RBP が提供する API 等が含まれている．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．プログラムの起動直後には何も格納されておらず，値は null である．

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，リカバリで復元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 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て，NVM アロケータの初期化と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．

4. リカバリ処理の詳細設計

本節では，3.2 節で示したリカバリ API である recovery メソッドについて，詳細な設計を記す．

4.1 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 7 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 への転送ポインタをセットする．GC が dram copy

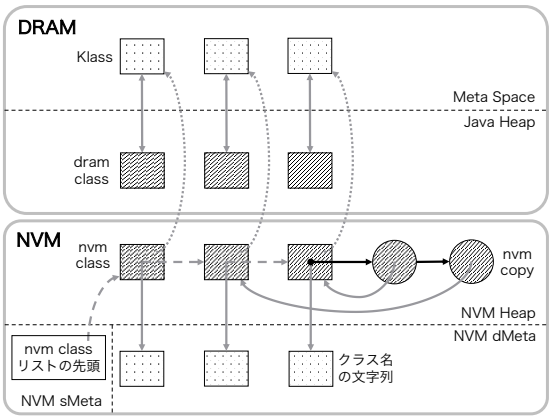

図 9 クラスのロード後のメモリの状態

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．

4.2 詳細設計

4.2.1 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 9 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．

4.2.2 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 10 の上部ある灰色の四角形は，参照型の配列オブジェクトを示している．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．

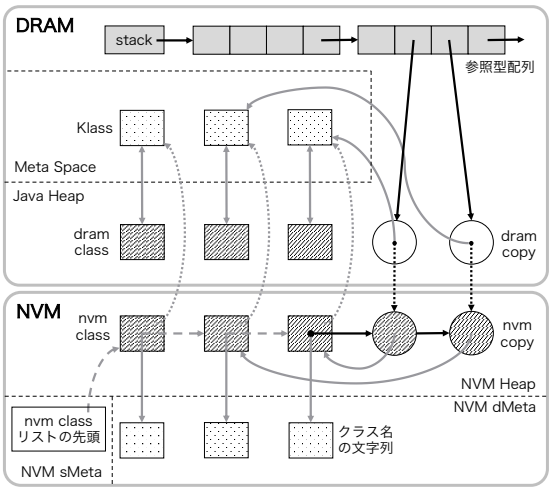

図 10 dram copy 領域を確保した後のメモリの状態

4.2.3 アロケータの復元

NVM アロケータの復元とは，メモリ管理用の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とである．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．

クラス情報は nvm copy のオブジェクトサイズを得るために必要である．メモリ管理用の DRAM 上のメタデータはクラッシュによって消えているため，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．

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．復元中に 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 を上書きしてしまう可能性があるため，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．

ここからは復元の実際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．また，その区間で最初に記録される nvm copy であったときは，その区間がアロケーションすべきサイズクラスも合わせて保存する．すべての 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復元が完了する．

特に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いるため，同様の手順でメタデータの復元が可能である．

4.2.4 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．

4.2.5 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．

4.2.5 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 4.2.4 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．

4.2.4 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．

4.3 リカバリ中のクラッシュへの対処

4.2.2 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．

4.2.2 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．

5. 評価

5.1 動作確認

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（5.2 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．

5.2 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

```
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) {
19       generator = new PrimeGenerator();
20     }
21
22     // アプリケーションの実行
23     generator.generate();
24   }
25 }
```

図 11 Primes クラス

```
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 = 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 12 PrimeGenerator クラス

Java アプリケーションの実装例について記す.

図 11 と図 12 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.

5.2.1 Primes クラス

図 11 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，19 行目で新たに PrimeGenerator のオブジェクトを作成する.

その後，23 行目でアプリケーションを実行する．この処理の詳細は 5.2.2 節で示す.

5.2.2 PrimeGenerator クラス

図 12 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 11 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，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ことを考慮して適切に行わなければならない．ここでは，既に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．

5.3 リカバリ時間

5.1 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．

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

Java VM には OpenJDK 16 を使用する．JIT コンパイラは無効化しており，インタプリタのみで動作する．また，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．

図 13 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．4.2.1 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 4.2.2 節，NVM アロケータのリカバリは 4.2.3 節，参照値変換の準備は 4.2.4 節，フィールドのコピーと nvm copy ヘッダのクリアは 4.2.5 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．

6. 関連研究

NVM は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 [2][8] 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

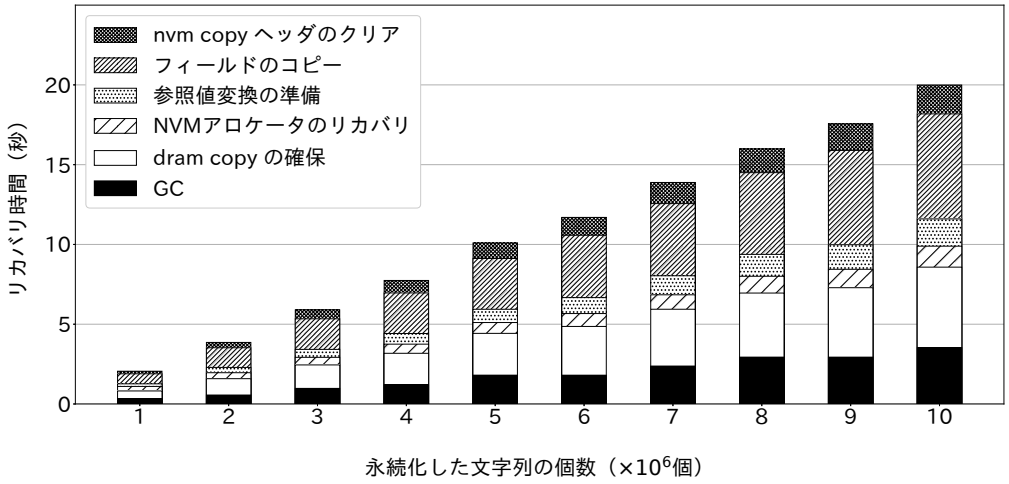

図 13 リカバリ時間と内訳

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 [2][8].

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 [2][3][4] がある一方，Espresso [8] 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 [9] 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 [2] 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，4.2.4 節と 4.2.5 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．5.3 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．

7. おわりに

本研究では，我々が開発している永続化機構を備えた Java VM で，クラッシュ後に永続データを

使用してプログラムを再開する仕組みを提供するために，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を提案した．

リカバリ機構を実現するために，クラス名などのメタ情報や永続ルートを NVM 上に追加で永続化した．リカバリは，DRAM 上にオブジェクトの領域を確保して，その内容を NVM 上の複製から書き戻す．リカバリ中のクラッシュや GC によるオブジェクトの移動と回収に注意しなければならない．これらを，ランタイムシステムに C++で実装した関数と Java のライブラリメソッドが協調して行うことで，複雑な処理を簡略化するよう工夫した．

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．また，提案システムを用いた 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．

謝辞 本研究の一部は，JSPS 科研費 19K11904 と 22H03566 の助成を受けたものです．

参考文献

[1] Atkinson, M. P. and Morrison, R.: Orthogonally Persistent Object Systems, *VLDB J.*, Vol. 4, No. 3, pp. 319–401 (online), available from ⟨https://doi.org/10.1007/BF01231642⟩ (1995).

[2] Cohen, N., Aksun, D. T. and Larus, J. R.: Object-Oriented Recovery for Non-Volatile Memory, *Proc. ACM Program. Lang.*, Vol. 2, No. OOPSLA (online), DOI: 10.1145/3276523 (2018).

[3] Friedman, M., Petrank, E. and Ramalhete, P.: Mirror: making lock-free data structures persistent, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*, ACM, pp. 1218–1232 (online), DOI: 10.1145/3453483.3454105 (2021).

[4] Haria, S., Hill, M. D. and Swift, M. M.: MOD: Minimally Ordered Durable Datastructures for Persistent Memory, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, New York, NY, USA, Association for Computing Machinery, p. 775–788 (online), DOI: 10.1145/3373376.3378472 (2020).

[5] James Gosling, Bill Joy, G. S. G. B. A. B. D. S. and Bierman, G.: The Java® Language Specification Java SE 16 Edition, `https://docs.oracle.com/javase/specs/jls/se16/html/index.html` (2021).

[6] Matsumoto, K., Ugawa, T. and Iwasaki, H.: Replication-Based Object Persistence by Reachability, *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2022, New York, NY, USA, Association for Computing Machinery, p. 43–56 (online), DOI: 10.1145/3520263.3534653 (2022).

[7] Tim Lindholm, Frank Yellin, G. B. A. B. and Smith, D.: The Java® Virtual Machine Specification Java SE 16 Edition, `https://docs.oracle.com/javase/specs/jvms/se16/html/index.html` (2021).

[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 Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018)*, ACM, pp. 70–83 (online), DOI: 10.1145/3173162.3173201 (2018).

[9] Wu, Z., Lu, K., Nisbet, A., Zhang, W. and Luján, M.: PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies, *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, ACM, pp. 623–637 (online), DOI: 10.1145/3385412.3386000 (2020).