オブジェクトの到達可能性による永続化をリードバリアを使わずに実現するアルゴリズムとその予備評価

鵜川 始陽, 松本 康太郎, 岩崎 英哉

到達可能性による永続化は，DRAM と不揮発性メモリ (NVM) を併用するシステムのマネージド言語に適した抽象化のひとつである．この抽象化では，オブジェクトは特定の変数から到達可能になった時に永続化される．これを実現する既存のアルゴリズムは，リードバリアを使って永続化されたオブジェクトにアクセスするため，読み出しとポインタの比較にオーバヘッドがかかる．本研究では，リードバリアを使わずに到達可能性による永続化を実現するアルゴリズムを提案する．提案アルゴリズムでは，DRAM 上のオブジェクトを NVM に複製することで永続化する．プログラムはリードバリアなしに DRAM のオブジェクトを使い続ける一方，書込みでは複製も更新する．複製は書込みバリアによって他のスレッドと非同期に作られるので，並行実行より生じる競合に対処したアルゴリズムになっている．このアルゴリズムを HotSpot VM に実装して評価した．

1 はじめに

近年，不揮発性メモリ (NVM) が登場したことにより，メモリ中のデータを小さいコストで永続化できるようになった．NVM 登場以前のシステムでは，アプリケーションは OS の力を借りて，永続化したいデータをディスクなどの二次記憶に送るか，ネットワークで他のシステムに送る必要があった．NVM は，DRAM 同様にバイト単位でアクセスでき，主記憶として DRAM と併用できる不揮発性の記憶装置である．アプリケーションプログラムは，データを NVM がマップされた仮想アドレスに配置するだけで，そのデータを永続化できる．つまり，システムの電源が失われるなどのシステムクラッシュがあっても，そのデータを保持することができる．

しかし，現状では，NVM のアクセス速度は DRAM と比べると遅い．Friedman ら [15] によれば読出しは DRAM の 3 倍遅い．さらに，アプリケーションは永続するデータを書き込むと，適切なタイミングでキャッシュラインを明示的にフラッシュし，NVM 上のデータが不整合なタイミングが存在しないようにしなければならない．このキャッシュフラッシュの時間も含めると，書込みには最大で DRAM の 588 倍の時間がかかる [33]．そのため，既存の NVM を使ったフレームワークでは，一般に NVM と DRAM を併用し，重要なデータだけを NVM に配置するアプローチをとっている．

永続化するデータを選択する方法や，どの時点のデータが永続化されるかは，フレームーワークによって異なる．例えば，C++用のデファクトスタンダードのフレームワークである PMDK[†1] ではプログラマが明示的に指定したクラスのインスタンスか，明示的に指定したオブジェクトを永続化する．また，適切にキャッシュラインをフラッシュするのはプログラマの責任としている．

一方で，近年，Java のようなマネージド言語に適したフレームワークが開発されている．この中には，到達可能性に基く永続化モデル [6] を採用しているフ

---

* This is an unrefreed paper. Copyrights blong to the Authors.

Tomoharu Ugawa, 東京大学, The University of Tokyo.
Kotaro Matsumoto, 高知工科大学, Kochi University of Technology.
Hideya Iwasaki, 電気通信大学, The University of Electro-Communications.

†1 https://pmem.io/pmdk/. 古いバージョンは NVML [22] と呼ばれていた.

レームワークがある．到達可能性に基づく永続化モデルでは，オブジェクトを連結して作ったデータ構造をまとめて永続化することができる．プログラマは永続化したいデータ構造のルートとなるオブジェクトが格納された static 変数を指定する．これを *durable roots* と呼ぶ．ランタイムシステムは，durable roots から到達可能なデータが常に永続化されていることを保証するように，自動的に必要な動作を行う．このモデルには，他の永続化モデルに比べて二つの利点がある．まず，プログラミングが簡単になる．さらに，既存のライブラリが内部で作るデータも必要に応じて自動的に永続化される．

しかし，このモデルを実現するための既存のアルゴリズムは大きなオーバヘッドを伴う．AutoPersist [24] は，このモデルを Java に実装した state of the art である．AutoPersist では，このモデルの実現のために，DRAM 上のオブジェクトが durable roots から到達可能になると，直ちにそのオブジェクトを NVM に移動させるアプローチをとっている．つまり，DRAM 上のあるオブジェクト o_1 へのポインタを NVM 上のオブジェクト o_2 のフィールドに書き込もうとすると，書込みバリアによって，o_1 と，o_1 から到達可能な全てのオブジェクトを NVM にコピーする．そして，DRAM 上にあるコピーしたオブジェクトのコピー元には，コピー先を指すフォワーディングポインタを残す．以降，コピーされたオブジェクトへのアクセスはリードバリアによって NVM 上のコピーへのアクセスとなる．次のごみ集め（GC）によって，コピーされたオブジェクトへの参照が NVM 上のコピーへの参照で置き換えられ，間接参照は解消されるが，アクセスしようとしているオブジェクトがフォワーデイングポインタを持つかどうかを調べるリードバリアは常に必要になる．また，二つの参照が同じオブジェクトを指すかどうかを調べる操作（==演算子）でも，フォワーディングポインタを調べる必要がある．これらのオーバヘッドは，オブジェクトを全く永続化しないプログラムにもかかる．さらに，永続化されたオブジェクトからの読出しは NVM からの読出しとなり，DRAM 上のオブジェクトよりも遅い．

本研究では，到達可能性に基づく永続化を実現するための新しいアルゴリズムである，複製永続化を提案する．このアルゴリズムは，AutoPersist に比べ，リードバリアが不要であり，永続化されたオブジェクトも DRAM から読み出すという利点がある．複製永続化は，データレースのないプログラムを想定している．そのようなプログラムに対して，durable root から到達可能な全てのオブジェクトが永続化され，かつ，永続化されたオブジェクトが，各スレッドの現在実行中か，直前の書込みのいずれかの結果を反映していることを保証する．データレースのないプログラムであっても，永続化している最中のオブジェクトに他のスレッドが書込む可能性はあり，その解消は自明ではない．提案アルゴリズムは，起こり得る競合を解消している．提案したアルゴリズムを OpenJDK の HotSpot VM のインタプリタに実装し，正しく動作することを確認した．

2 準備

2.1 用語の定義

本節では，以降で用いる用語を定義する．**オブジェクト**とは，プログラムの視点でのデータ構造を指す．例えば，

```
C x = new C(...);
```

によって，変数 `x` はクラス `C` のコンストラクタによって作られたオブジェクトを持つ．永続化されていないオブジェクトは**揮発性オブジェクト**，永続化されたオブジェクトは**永続化オブジェクト**と呼ぶ．

ランタイムシステムは，オブジェクトのデータをメモリ上に割り当てる．DRAM 上に割り当てられたオブジェクトのデータは**揮発性コピー**，NVM 上に割り当てられたデータは**永続化コピー**と呼ぶ．ただし，あいまいでない時は，揮発性コピーや永続化コピーを揮発性オブジェクトや永続化オブジェクトと呼ぶことがある．疑似コードでは，揮発性コピーと永続化コピーをそれぞれ `VCopy` 型と `PCopy` 型の変数で表す．

2.2 キャッシュラインフラッシュ

NVM に格納されたデータは電源が失われても保存されるが，NVM がマップされた仮想アドレスに書き込んだデータが直ちに永続化される訳ではない．書

込みは CPU のキャッシュメモリを介して行われるからである．キャッシュメモリは揮発性であり，電源が失われるとダーティであるキャッシュラインのデータは失われる．書込みを永続化するには，その書込みを行ったキャッシュラインのデータを NVM に書き戻す必要がある．

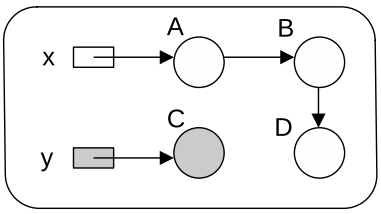
Intel x64 アーキテクチャ [18] の CPU でキャッシュラインを書き戻すには，`CLWB` 命令が利用できる．この命令は，指定したキャッシュラインを無効化することなく書き戻す．キャッシュラインを無効化するかどうかはアルゴリズムの正しさには影響しないので，以降では，キャッシュラインの無効化に関係なく，キャッシュラインを書き戻すことを，キャッシュラインを**フラッシュ**すると言う．

`CLWB` 命令はキャッシュフラッシュが完了するのを待つことなく次の命令を実行する．キャッシュフラッシュの完了を保証するためには，`SFENCE` などのメモリバリア命令を実行する必要がある．メモリバリア命令の完了によって，先行する全ての `CLWB` 命令によるキャッシュフラッシュが完了したことが保証される．より正確には，`CLWB` 命令によるキャッシュラインフラッシュは，不可分な compare-and-swap に用いるロックプレフィックス付きの命令などのストアフェンスの機能を持つ命令との間で順序が保証される．また，同じアドレスに対する書込みとも順序が保証される．

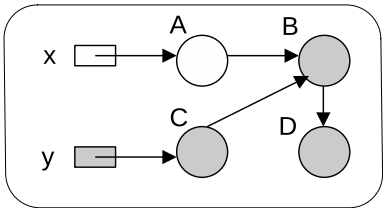
2.3 到達可能性に基づく永続化

到達可能性に基く永続化モデルには，第 1 章で示したような利点があり，マネージド言語に適したモデルである．特に，プログラマは個々のオブジェクトが DRAM にあるか NVM にあるかを気にすることなくプログラミングすることができる．プログラマは，オブジェクトを作る時にそのオブジェクトを永続化するかどうかを指定する必要はない．全てのオブジェクトは，作られた時は DRAM 上の揮発性コピーだけを持つ．その後，そのオブジェクトが durable roots から到達可能になった時に，タンタイムシステムが自動的に永続化する．

ランタイムシスムは，durable roots や永続化オブジェクトが揮発性オブジェクトを参照することがないことを保証する．これによって，NVM 上のデータから復元する際に，NVM 上の全てのオブジェクトの参照先が NVM 上に存在することが保証される．



(a) C から B への参照の書込み前.



(b) C から B への参照の書込み後.

図 1　到達可能性に基づく永続化モデル.

例えば，オブジェクト A, B, C, D が図 1 (a) のように接続されている場合を考える．ここで x と y はプログラム上の変数であり，灰色の変数とオブジェクトは永続化されていることを表す．この例では，y が durable roots として指定されており，それにより C が永続化されている．ここで，C から B への参照が作られたとする．このとき，ランタイムシステムは B だけでなく，D も永続化し，図 1 (b) のようになる．D が B から到達可能だからである．

3 設計

この章では，提案アルゴリズムを概観し，シングルスレッド向けに簡単化したアルゴリズムを説明する．その後，第 4 章でマルチスレッド向けに拡張する．

3.1 永続化モデル

再起動後，システムのクラッシュ前のデータを正しく復元するためには，直観的には，NVM 上のデータが整合性のとれた状態を保つ必要がある．しかし，NVM に書き込まれる順序がプログラム順序と異なる可能性があると，NVM 上のデータの整合性がとれた

状態を保つのは難しい．

そこで，本研究では，先行研究である AutoPersist [24] と同様に，永続化オブジェクトへの書込みの結果が，その書込みが完了するまでに NVM に書き込まれることを保証する．これにより，同じスレッドが実行する複数の書込みは，プログラム順序で永続化される．

一方で，複数のスレッドが排他制御せずに同じフィールドにアクセスする場合は，この保証では不十分なことがある．あるスレッド T_1 の書込み w_1 が永続化される前に，その値を別のスレッド T_2 が読み出し，スレッド T_2 は読み出した値に依存した書込み w_2 を行ったとする．このとき，w_2 は w_1 より早く永続化される可能性がある．例えば，スレッド `T1` と `T2` が次のようなプログラムを並行に実行したとする．

```
T1: x.v = 1
T2: if (x.v == 1) y.v = 1;
```

このとき，`x.v = 1` が行われて，その結果が永続化されないうちに，T2 はキャッシュメモリから `x.v` の値を読み出すことが可能である．その結果，`y.v = 1` が行われて，その結果が先に永続化されることがある．そこでシステムがクラッシュすると，復元した時に `x.v == 0 && y.v == 1` ということが起こり得る．

本研究では，データレースのないプログラムのみを対象とし，このような問題は取り扱わない．なお，AutoPersist では，トランザクションの全ての書込みがコミット時に不可分に永続化される永続化トランザクションの機能も提供しており，同じフィールドへのアクセスは，従来のモニタロックを用いるか，永続化トランザクションを用いて排他制御することになっている．永続化トランザクションは本研究の提案とは直交しており，対象としない．

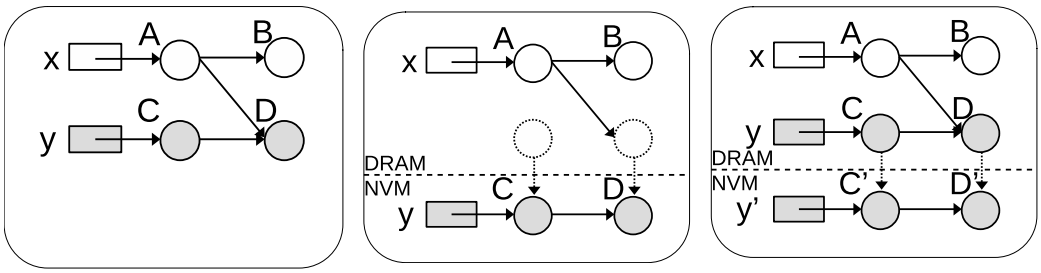
3.2 基本的なアイデア

我々は，到達可能性に基づく永続化を実現する新しいアルゴリズムとして，複製永続化を提案する．このアルゴリズムはリードバリアを用いず，かつ，永続化されたオブジェクトからの読み出しに NVM アクセスを伴わない．

先行研究である AutoPersist [24] では，オブジェクトが永続化される時，そのオブジェクトを NVM に**移動**させる．具体的には，そのオブジェクトの永続化コピーを作り，揮発性コピーの領域に永続化コピーへのフォワーディングポインタを上書きする．その結果，オブジェクトは常に揮発性コピーか永続化コピーのいずれか一方を持つことになる．

例えば，オブジェクト A, B, C, D が図 2 (a) のように接続されているとする．ここで x と y は変数である．また，灰色は永続化されていることを表す．つまり，y は durable roots であり，C と D は永続化オブジェクトである．AutoPersist では，この時の内部実装は図 2 (b) のようになっている．ここで C や D の移動前の位置は点線の丸で表している．これらは C や D へのフォワーディングポインタを持っており，移動前の C や D へのポインタを使ったアクセスがあると，リードバリアや書込みバリアによって移動先に対してアクセスされる．さらに，二つの参照が等しいか調べる操作（Java では `if_acmpeq` 命令）でも，一方の参照が移動前を，他方が移動先を指すポインタである可能性があるので，適切なバリアが必要である．GC が起こると，A は直接 D を指すようになるが，それでもオブジェクトが移動していないかどうか調べるためのリードバリアは必要である．また，永続化したオブジェクトは永続化コピーしか持たないため，永続化したオブジェクトからの読出しは NVM からの読出しになる．

それに対して，提案する複製永続化はオブジェクトを複製することによって永続化する．複製永続化では，オブジェクトを永続化しても揮発性コピーを保持し続ける．つまり，永続化オブジェクトは揮発性コピーと永続化コピーの両方を持つ．揮発性コピーをフォワーディングポインタで上書きするのを避けるため，オブジェクトヘッダ内に専用に設けた `pcopy` フィールドにフォワーディングポインタを書き込む．揮発性コピーと永続化コピーは同じ**内容**を保持するように，書込みバリアによって同期する．同じ内容を保持するとは，プリミティブ型のフィールドは同じ値を保持し，参照型のフィールドに対しては揮発性コピーと永続化コピーがそれぞれ同じオブジェクトの揮発性コピーと永続化コピーを指すポインタを保持す



(a) プログラマの視点. (b) AutoPersist. (c) 複製永続化.

図 2　永続化のアプローチ.

ることを言う.

複製永続化では，図 2 (a) のヒープの内部実装は図 2 (c) のようになる．全てのオブジェクトは DRAM 上の揮発性コピーを持つ．さらに，永続化オブジェクトは NVM 上の永続化コピーも持つ．図では，C と D が揮発性コピーであり，C' と D' がそれぞれ対応する永続化コピーである．揮発性コピーは `pcopy` フィールドに永続化コピーへのポインタを持つ.

Java プログラムの参照は常に揮発性コピーへのポインタで表現される．つまり，参照を持つ変数は揮発性コピーへのポインタか `null` を持つ．ただし，durable roots として指定された static 変数は例外で，そのような変数は DRAM と NVM に領域を持ち，揮発性コピーと永続化コピーの両方を持つ.

アプリケーションがオブジェクトから読み出す時は，永続化されているかどうかに関係なく揮発性コピーから読み出す．オブジェクトへの参照が揮発性コピーへのポインタで表現されているので，リードバリアは必要ない．一方，永続化オブジェクトに書き込む時は，揮発性コピーと永続化コピーを同期させるために，書込みバリアによって両方に書き込む．この両方に書き込む処理を double update と呼ぶ．double update の間には，一方だけに書き込まれた瞬間が存在するが，データレースがないプログラムを仮定しているので，この瞬間を他のスレッドが読む心配はない.

3.3　オブジェクトの色と不変条件

オブジェクトを永続化する処理の説明のために，以下の色でオブジェクトの状態を表すことにする.

1. 白オブジェクトは永続化コピーを持たない.
2. 灰色オブジェクトは永続化コピーの領域は持つが，その内容は揮発性コピーと同期していない．灰色オブジェクトから直接指されるオブジェクトは，永続化コピーを持たないかもしれない.
3. 黒オブジェクトは揮発性コピーと同期された永続化コピーを持つ．永続化コピーの存在するキャッシュラインに対して `CLWB` 命令も発行しているが，キャッシュラインのフラッシュは完了していないかもしれない.
4. 青オブジェクトはキャッシュフラッシュが完了している．さらに，青オブジェクトから指されるオブジェクトも青である.

白，灰色，黒の 3 色は，Dijkstra による 3 色の抽象化 [14] と似ているが，永続化に関する条件も追加されている．青になったオブジェクトはクラッシュの後に復元できる．なお，色は説明のためのもので，オブジェクトは明示的に色を表すデータを持つことはない.

オブジェクトを複製する処理は，この 4 色を使って説明できる．白オブジェクトを複製するときは，そのオブジェクトの永続化コピーの領域を割り当てて，そのポインタを揮発性コピーの `pcopy` フィールドに書込む．これによってそのオブジェクトは灰色になる．次に，揮発性コピーの内容を永続化コピーにコピーする．このとき，このオブジェクトから指される子オブジェクトが白であれば，子オブジェクトにも永続化コピーの領域を割り当て，灰色にする．さらに，永続化コピーの領域に `CLWB` を発行することで，そのオブジェクトを黒にする．最後に，そのオブジェクトから到達可能な全てのオブジェクトのキャッシュフラッ

シュの完了を待って，オブジェクトは青，つまり復元可能な状態になる．

複製永続のアルゴリズムは，白，灰色，黒のオブジェクトに対して強い 3 色不変条件を満たす．つまり，

不変条件 1 黒オブジェクトが白オブジェクトを直接指すことはない．

また，青オブジェトに関して，

不変条件 2 青オブジェクトが青以外のオブジェクトを指すことはない．

これに加えて，次の二つの不変条件も満たす．

不変条件 3 黒か青のオブジェクトでは，書込みバリアの実行中を除いて，揮発性コピーと永続化コピーの内容が一致する．

不変条件 4 青オブジェクトでは，書込みバリアの実行中を除いて，永続化コピーの内容が *NVM* に書込まれている．

不変条件 3 と 4 より，システムがクラッシュした後でも，クラッシュの時にちょうど書込みバリアを実行していたオブジェクトを除いて，青オブジェクトについては，揮発性コピーと同じ内容の永続化コピーが NVM 上に存在する．システムがクラッシュした時にちょうど書込みバリアが実行されていたオブジェクトについては，揮発性コピーだけが更新されている可能性がある．そのような場合は，書込み前にクラッシュしたとみなす．データレースのないプログラムでは，揮発性コピーだけが更新された状態が他のスレッドから観測されておらず，書込みの前にクラッシュしたとみなしても NVM のデータの一貫性が崩れることはない．さらに，不変条件 2 より，NVM 上のオブジェクトの参照先は NVM に存在することが保証できる．

3.4 シングルスレッド版永続化アルゴリズム

まず，簡単のために，シングルスレッドのプログラムを仮定して永続化のアルゴリズムを説明する．図 3 に永続化処理の関数 `ensure_recoverable` を示す．ここで，オブジェクト `o` のフィールド `f` を `o[f]` と書く．簡単のために，全てのオブジェクトは，ただ一つの参照型のフィールドを持つとする．シングルスレッドプログラムを仮定しているので，`ensure_recoverable` が呼び出されたとき，永続化の最中のオブジェクトは

```
/* make r persistent */
ensure_recoverable(VCopy r) {
  if (shade(r) == TRUE) {
    /* r is gray */
    worklist.add(r);
    while (not worklist.empty()) {
      VCopy o = worklist.remove();
      PCopy on = o.pcopy;
      /* Object o is gray */
      foreach(Field f in o.fields) {
        VCopy p = o[f];
        if (shade(p) == TRUE)
          worklist.add(p);
        on[f] = p.pcopy;
      }
      FLUSH_RANGE(on, on.size());
      /* Object o is black */
    }
    SFENCE;
  }
}

/* If o is white, shade(o) colors o gray and
     returns TRUE */
shade(VCopy o) {
  if (o.pcopy != NULL) /* o is not white */
    return FALSE;
  /* Allocate a persistent copy */
  o.pcopy = alloc_NVM(o.size());
  return TRUE;
}
```

図 3　シングルスレッド版 `ensure_recoverable`.

存在しない．つまり，全てのオブジェクトは白か青である．

永続化するオブジェクトの揮発性コピーへのポインタを `r` とする．`ensure_recoverable` は，まず，`shade` を呼び出して，`r` で指されるオブジェクトが白であれば灰色にする．この処理は，`r` に永続化コピーの領域を割り当て，そのポインタを揮発性コピーの `pcopy` フィールドに書き込むことにより行われる．

その後，灰色オブジェクトを管理するワークリスト `worklist` を使って，`r` から到達できる白オブジェクトを再帰的にたどり，灰色を経由して黒に変える．具体的には，ワークリストから灰色オブジェクト `o` を取り出し，

1. `o` のフィールド `f` に書き込まれたポインタ `p` を `shade` 関数に渡して `p` で指されるオブジェクトを灰色にする．
2. `p` の永続化コピーへのポインタを `o` の永続化コピーのフィールド `f` に書込む．

```
putfield(VCopy o, Field f, VCopy v) {
  PCopy on = o.pcopy;
  if (on == NULL) {
    /* o is white */
    o[f] = v;
    return;
  }
  /* o is blue */
  ensure_recoverable(v);
  o[f] = v;
  on[f] = v.pcopy;
  FLUSH(&on[f]);
  SFENCE;
}
```

図 4　シングルスレッド版 putfield.

o の全てのフィールドを処理し終えたら，o の永続化コピーが配置されている全てのキャッシュラインのフラッシュを開始する．フラッシュの完了は，ここでは待たない．この時点で，o は黒になる．

最後に，r から到達できる全ての白だったオブジェクトが黒になると，SFENCE 命令を発行し，これらの黒オブジェクトを同時に青にする．

3.5　書込みバリア

図 4 にシングルスレッド版の書込みバリアを示す．putfield は v を o のフィールド f に書き込む関数である．putfield では，3.3 節で示した不変条件を守るために，二種類の書込みバリアを用いる．

まず，不変条件 2 を守るために，挿入バリア [14] を用いる．シングルスレッドプログラムを仮定しているので，o の pcopy フィールドが NULL でなければ o は青である．青オブジェクトに書き込む時は，書き込む値 v が青であるよう，書込み前に ensure_recoverable 関数で v を青に変える．

次に，不変条件 3 を守るために，揮発性コピーと永続化コピーの両方に書き込む double update バリアを用いる．o のフィールド f に v を書き込む時は，同時に，v の永続化コピーへのポインタ v.pcopy を o の永続化コピーのフィールド f に書き込む．

さらに，不変条件 4 を守るために，永続化コピーの書き込んだフィールドに対してキャッシュフラッシュを発行し，それが完了するのを SFENCE で待つ．永続化オブジェクトに書込むたびにキャッシュフラッシュと SFENCE を実行するとオーバヘッドが大きいが，これは必要なコストである．なお，既存研究である AutoPersist [23] [24] でも同じ処理を行っている．

4　並列実行

第 3 章で示したアルゴリズムは，マルチスレッド環境では正しく動作しない可能性がある．本アルゴリズムは，データレースのないプログラムを仮定しているが，それでも，マルチスレッド環境では以下に挙げる競合が起こる可能性があるからである．

1. あるスレッドが putfield である白オブジェクト o に書き込んでいる最中に，別のスレッドが o を永続化する．
2. あるスレッドがオブジェクト o を永続化している最中に，別のスレッドが putfield で o に書込む．
3. 複数のスレッドが同時に同じオブジェクト o を永続化する．

プログラマは注意深くプログラミングすることで，プログラム上のデータレースは避けることができる．一方，オブジェクトの永続化はランタイムシステムが自動的に行うため，オブジェクトを永続化する処理が関連する競合はプログラマが避けることはできない．これは，我々の目指すマネージドランタイムに適した永続化モデルでは，オブジェクトの永続化状態はプログラマが気にする必要がないためである．例えば，上記 3 の競合は，複数のスレッドが o や青でないオブジェクトだけをたどって o に到達できるオブジェクトの参照を，同時に永続化オブジェクトに書込むことで起こる．この競合はプログラマには制御できず，ランタイムシステムが解決する必要がある．

この章では，上記の競合条件を，Intel x86 などの total store ordering (TSO) [1] のメモリ一貫性を持つ CPU でどのように解決するかを示す．

4.1　オブジェクトを永続化するスレッド間の依存関係

ensure_recoverable 関数（図 3）は，引数の r から青でないオブジェクトだけをたどって到達できる全てのオブジェクトを青にする．この，r から青でない

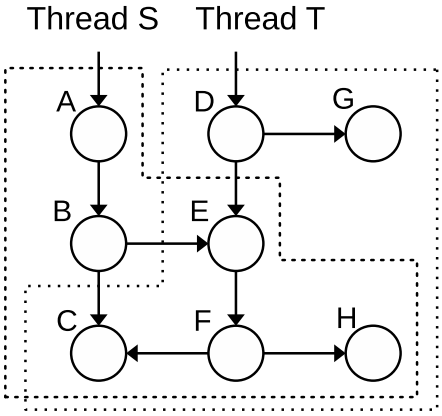


図 5　PS 集合.

```
shade(VCopy o) {
  PCopy on = alloc_NVM(o.size());
  PCopy fwd;
  while ((fwd = CAS(&o.pcopy, NULL, on)) == BUSY
     )
    ;
  if (fwd != NULL) {
    /* o is gray, black, or blue */
    t = responsible_thread(fwd);
    if (t != current_thread) {
      /* thread t is making o recoverable */
      depends_on(t);
    }
    return FALSE;
  }
  return TRUE;
}
```

図 6　マルチスレッド版 shade.

オブジェクトだけをたどって到達できるオブジェクトの集合を ensure_recoverable を呼び出したスレッドの**永続化対象集合**（persistent subject 集合; PS 集合）と呼ぶことにする．複数のスレッドが並行して ensure_recoverable 関数を呼び出すと，これらの関数の PS 集合が重なる可能性がある．PS 集合が重なるスレッドの関係の推移閉包は同値関係になる．この同値関係による同値類を**干渉スレッドグループ** (interfering thread group; ITG) と呼ぶことにする．

例えば，図 5 のような状況を考える．ここで，スレッド S と T がそれぞれ A と D を引数に ensure_recoverable を呼び出したとする．このとき，S の PS 集合は $\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{E},\mathsf{F},\mathsf{H}\}$ であり，T の PS 集合は $\{\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F},\mathsf{G},\mathsf{H}\}$ である．二つの PS 集合には重なりがあるので，S と T は同じ ITG に属する．

複数のスレッドの PS 集合に重なりがあるとき，これらのスレッドは同じオブジェクト o を永続化しようとする．この競合を解決するために，o にフォワーディングポインタを書込む操作に不可分な compare-and-swap を使い，これに成功したスレッドが o を黒にすることにする．このスレッドを o の**担当スレッド**と呼ぶ．

図 6 にマルチスレッド版の shade 関数を示す．もし永続化対象オブジェクト o の pcopy フィールドが NULL であれば，o は白である．このスレッドは，永続化コピーの領域を割り当てて，そのポインタを不可分な compare-and-swap（CAS）命令で pcopy フィールドに書き込む．ここで，CAS は第一引数のアドレスの値が第二引数の値と一致すれば，不可分に第三引数の値で更新する関数とし，戻り値は更新の成否によらず，関数実行前に第一引数のアドレスに書かれていた値とする．CAS の戻り値が NULL であれば，CAS が成功したことを表す．このとき，o は灰色になり，このスレッドが o の担当スレッドになる．図 6 では，担当スレッドは responsible_thread 関数で得られるとしている．responsible_thread の実装は第 5 章で述べる．

CAS が失敗した場合，以下の状況が考えられる．

- オブジェクト o が CAS を実行しているスレッド自身によって既に灰色にされていた場合．オブジェクトグラフが循環を含むような場合に，このようなことが起こる．o が既に灰色になっており，カレントスレッドが担当スレッドになっているので，加えて何かする必要はない．
- オブジェクト o が他のスレッド t によって灰色や黒になっている場合．この場合は，担当スレッドは t である．depends_on 関数によって，カレントスレッドと t が同じ ITG に属することを記録する．
- o が既に青になっている場合．この場合は何もする必要がない．
- 他のスレッドが putfield によって o に書き込んでいる最中の場合．この場合，4.3 節で後述す

るように，o の pcopy フィールドには特別な値 BUSY が書込まれている．カレントスレッドは書込みが終わって BUSY がクリアされるまで待つ．詳細は 4.3 節で述べる．

4.2 同一 ITG に属するスレッド間のバリア同期

スレッド T の PS 集合に，別のスレッド S が担当スレッドであるオブジェクトが属するとき，T が S に依存しているという．不変条件 2 を守るためには，T がスレッド S に依存しているとき，S が S の PS セットを青にするまでは，T は T の PS セットを青にできない．しかし，同一 ITG に属するスレッドは，互いに依存する可能性がある．例えば，図 5 で，C と E の担当スレッドがそれぞれ S と T だったとする．このとき，C について T は S に依存する．一方で，E について S は T に依存する．

このような場合には，S と T が同時にそれぞれの PS 集合を青にする．そのために，各スレッドは PS 集合を黒にし，SFENCE を実行した後，同一 ITG に属するスレッドとバリア同期をとる．このバリア同期の完了をもって，この ITG に属するスレッドの PS セットに属する全てのオブジェクトは同時に青になる．

図 7 はマルチスレッド版の ensure_recoverable である．この関数の最後でバリア同期（SYNC）を実行している．SYNC 以外のシングルスレッド版との違いは#の印の行である．この違いの詳細は 4.4 節で述べる．

4.3 白オブジェクトへの書込み

あるスレッド T が白オブジェクト o に書き込んでいる最中に別のスレッド S が o を永続化しようとする可能性がある．T は o が白から灰色や黒に変わったことに気づかなければ，T は double update バリアを実行せず，不変条件 3 に反して o の揮発性コピーと永続化コピーに不整合が生じる．さらに，スレッド T は白オブジェクトへのポインタを黒や青に変わったオブジェクト o に書き込んでしまうかもしれない．その結果，不変条件 2 が崩れることになる．このような問題を防ぐために，白オブジェクトに書き込もうとするスレッドは，オブジェクトが白であることを確認すると同時に，そのオブジェクトが灰色にならないようにロックをかける．

```
ensure_recoverable(VCopy r) {
  if (shade(r) == TRUE) {
    worklist.add(r);
    while (not worklist.empty()) {
      VCopy o = worklist.remove();
      PCopy on = o.pcopy;
      /* o is gray */
#   RETRY:
      /* copy */
      for (Field f in o.fields) {
        VCopy p = o[f];
        if (shade(p) == TRUE)
          worklist.add(p);
        on[f] = p.pcopy;
      }
#     MFENCE;
      CLWB_RANGE(on, on.size());
#     /* verify */
#     foreach (Field f in o.fields)
#       if (o[f].pcopy != on[f])
#         goto RETRY;
      /* complete — o is black */
    }
    SFENCE;
    /* We still don't know about the color of
     objects beyond the objects for which other
     threads are responsible. */
  }
  SYNC();
  /* r's transitive closure is blue */
}
```

図 7 マルチスレッド版 ensure_recoverable.

図 8 にマルチスレッド版の putfield を示す．putfield を実行しているスレッドは書込み先のオブジェクト o に対して，o が白であれば pcopy フィールドに特別な値 BUSY を書込むことで，ロックをかける．o が白であるかどうかは pcopy が NULL かどうかで判定する．この判定と書込みは，不可分な compare-and-swap 命令により行う．他のスレッドが既にロックをとっていた場合は，CAS 命令は BUSY を返す．この場合，BUSY がクリアされるのを待つ．CAS 命令が BUSY でも NULL でもない値を返すと，o が白ではなかったことを意味する．このときは，3.5 節で述べたように，挿入バリアと double update バリアを実行する．

参照ではなくプリミティブ型の値を書き込む時はバリアを簡単にできる．不変条件 2 が崩れる心配がないからである．プリミティブ型の値を書き込む putfield

```
putfield(VCopy o, Field f, VCopy v) {
  PCopy on;
  while ((on = CAS(&o.pcopy, NULL, BUSY)) ==
     BUSY);
  if (on == NULL) {
    /* o is white */
    o[f] = v;
    o.pcopy = NULL;
    return;
  }
  /* o is gray, black, or blue */
  ensure_recoverable(v);
  PCopy vn = v.pcopy;
  o[f] = v;
  on[f] = vn;
  CLWB(&on[f]);  /* write back the cache line
     for o.f in persistent copy */
  SFENCE;
}
```

図 8　マルチスレッド版 putfield （参照用）.

```
putfield(VCopy o, Field f, Primitive v) {
  o[f] = v;
  MFENCE;
  PCopy on = o.pcopy;
  if (on == NULL || on == BUSY) {
    /* o is white */
    return;
  }
  /* o is gray, black, or blue */
  on[f] = v;
  CLWB(&on[f]);
  SFENCE;
}
```

図 9　マルチスレッド版 putfield （プリミティブ用）.

を図 9 に示す．この putfield は，まず無条件に v を o の揮発性コピーに書き込む．その後，pcopy フィールドを調べて，o が白かどうかを確認する．もし白であれば，白のオブジェクトの揮発性コピーに値を書き込んだことになっており，これ以上することはない．白でなければ，double update バリアを実行する．揮発性コピーへの書込みと，オブジェクトの色を調べる読出しとの間には，オブジェクトの色を調べる前に書込みが完了するように MFENCE を実行する．

4.4　コピーと書込みの競争

いま，スレッド T が o を引数に ensure_recoverable 関数を実行しており，その中で，o のフィールド f を o の永続化コピー on にコピーしているとする．このとき，別のスレッド S がほぼ同時に v を f に書き込んだとすると，以下のような順序で実行される可能性がある．

step	thread T	thread S
1:	p = o[f]	
2:		o[f] = v
3:		on[f] = v.pcopy
4:	on[f] = p.pcopy	

その結果，S によって NVM 上の on[f] に書かれた v は T によって上書きされ，失われてしまう．これにより，不変条件 3 が崩れる．

この競合は，並行コピー GC である Transactional Sapphire GC [28] で使われているコピープロトコルを用いて解決する．このプロトコルは，データレースのないプログラムに対して，TSO メモリ一貫性モデルの CPU で利用できる，高速に一貫性を保ちながらコピーするプロトコルである．図 7 の “#” の印を付けた行がこのプロトコルを実装している．

このコピープロトコルはコピーと検証の二つのステップからなる．コピーステップでは，通常のロードとストア命令を使ってナイーブに揮発性コピーから永続化コピーにフィールドの値をコピーする．この時，並行して他のスレッドが書き込んでいるかもしれない．検証ステップでは，他のスレッドの書込みがなかったか，あるいは影響のない書込みだったことを検証する．具体的には，検証ステップになっても，ensure_recoverable がコピーステップで読み出したのと同じ値が揮発性コピーから読み出せることを確認する．検証に失敗すれば，コピーステップに戻ってやり直す．コピーステップと検証ステップの間には MFENCE を実行し，書込みが完了する前に検証を始めるのを防ぐ．このプロトコルの正しさは，文献 [27] で検証している．

4.5　補助関数

複製永続化のアルゴリズムは，responsible_thread, depends_on, SYNC の三つの補助関数を用いる．ここでは，これらの補助関数が満たすべき条件を述べる．

responsible_thread 関数はオブジェクト o の永続化コピーのポインタを受けとり，o が灰色か黒であれば，その担当スレッドを返す．o が青であれば，NULL を返すことが期待される．しかし，o が青になるのは，o の担当スレッドと同じ ITG に属する最後のスレッドがバリア同期に到達した瞬間なので，responsible_thread がブロックせずに，そのタイミングを正確に知るのは難しい．そこで，responsible_thread は青のオブジェクトに対しては，NULL の他に，どんなスレッドを返してもよいことにする．

depends_on 関数は responsible_thread 関数が返した，o の担当スレッドと期待されるスレッド T を引数に受け取る．depends_on は，カレントスレッドが属する ITG と T が属する ITG を併合する．

ただし，depends_on が呼び出された時には，T は既に ensure_recoverable を抜けている可能性がある．より正確には，SYNC によるバリア同期を終えている可能性がある．depends_on 関数はこのようなスレッドを検出して，誤った処理をしないようにしなければならない．幸いなことに，ITG の仕組みを素直に実装すると，各スレッドはスレッド構造体に，どの ITG に属するかを示すフィールドを持つことになる．そのため，スレッドがいずれかの ITG に属するかどうか調べることで，ensure_recoverable の実行中かどうかを判定できる．また，ensure_recoverable を抜けたスレッドが別のオブジェクトの永続化のために再度 ensure_recoverable の実行を始めた場合，本来依存していないスレッド間でもバリア同期を行うことになるが，正しさに影響はない．

5 実装

複製永続化のアルゴリズムが正しく動作することを確認するために，我々は，OpenJDK の HotSpot VM のインタプリタに実装した．ここでは，実装の詳細を説明する．

5.1 担当スレッド

responsible_thread は，永続化コピーに担当スレッドへのポインタを持たせることで実現する．このポインタは，永続化コピーを作るときに設定し，そのオブジェクトが青になるとクリアする．

具体的には，shade 関数（図 6）内の alloc_NVM が永続化コピーに対して 2 ワード余分にメモリを割り当てる．この 2 ワードは，担当スレッドへのポインタを保持するためと，同じ担当スレッドを持つオブジェクトのリストを構成するために用いる．永続化コピーの領域を割り当てると，そのポインタを揮発性コピーに書き込む前に担当スレッドへのポインタを書き込む．これにより，永続化コピーへのポインタを揮発性コピーに書き込むことに成功すると同時に，このスレッドが担当スレッドになる．担当スレッドがバリア同期 (SYNC) を抜けると，リストに連結されたオブジェクトの担当スレッドポインタをクリアする．

5.2 Recoverable Bit

複製永続化のアルゴリズムでは，オブジェクトが青かどうかの判定は頻繁に行われる．青かどうかの判定は，(1) 揮発性コピーの pcopy フィールドが NULL でないことを確認し，さらに (2) 永続化コピーの担当スレッドのポインタが NULL であることを確認することで判定できるが，この判定には時間がかかる．そこで，各オブジェクトの揮発性コピーに青になるとセットされる *recoverable bit* を持たせる．NVM 上のオブジェクトは 8 バイトに整列されていることを利用して，recoverable bit には pcopy フィールドの最下位ビットを流用する．

recoverable bit がクリアされていても青である可能性はあるので，その場合は，詳細に調べる必要がある．しかし，recoverable bit がクリアされている場合，過渡的な場合を除いて，そのオブジェクトは白であり，pcopy フィールドが NULL である．そのため，ほとんどの場合は，pcopy を調べるだけで青かどうかを判定できる．

6 評価

第 5 章で示した実装を用いて，性能を評価した．この評価は，提案アルゴリズムが既存研究である AutoPersist より高速であることを定量的に示すためのものではなく，提案アルゴリズムに予想外のオーバヘッドがないことを確認するためのものである．この

章では，実験の結果に加えて，既存研究との定性的な比較を示す．既存研究との定量的な比較は今後の課題である．

6.1 Evaluation Settings

本研究では，OpenJDK 13 の JIT コンパイラを無効化した HotSpot VM を用いた．また，DRAM 領域の GC には stop-the-world の serial GC を用い，NVM 領域には GC は実装していない．しかし，NVM の領域は十分に大きく，ベンチマークプログラムの実行で NVM の領域が不足することはなかった．NVM のメモリ割り当てのためには，各スレッドには 50KB のスレッドローカルアロケーションバッファを持たせた．

実験環境の計算機には，128 GB の第一世代 Intel Optane DC persistent memory を 2 枚搭載した．CPU は Intel Xeon Gold 6240 (18 コア，36 スレッド，クロックは 2.60 GHz に固定) を 1 台搭載し，DRAM は 48 GB の DDR4 を搭載した．OS は Linux 4.8 Ubuntu 18.04.5 LTS を用いた．NVM は NOVA ファイルシステムでフォーマットし，アプリケーション DAX モードで `mmap` して用いた．HotSpot VM は “server” の設定でビルドした．この環境での初期ヒープサイズは 734 MB，最大ヒープサイズは 11 GB だった．

ベンチマークプログラムには DaCapo ベンチマーク集 [9] バージョン dacapo-9.12-MR1-bach を用いた．評価には，複製永続化を実装していない HotSpot VM でも実行できなかった batik, eclipse, tomcat を除く全てのベンチマークプログラムを用いた．各実行では十分にウォームアップし，各イテレーションの実行時間が収束するまで繰返した（`-C` オプション）．さらに，各ベンチマークを最低 5 回実行した．実験結果のグラフには，中央値と四分位点を示している．

表 1 に DaCapo ベンチマーク集のプログラムの基本的な性質を掲載する．time は，複製永続化を実装していない HotSpot VM での実行時間，pri はプリミティブ型の書込み頻度，ref は参照の書込み頻度である．

ベンチマークプログラムには durable roots の指定はなされていない．そこで，恣意的に durable roots を選ぶのを避けるために，durable roots を全く指定しない none-durable と，全ての static 変数を durable roots とする all-durable の二つの設定で実験した．

表 1　ベンチマクプログラムの基本データ．

name	time (s)	writes pri (M/s)	writes ref (M/s)
avrora	17.7	23.47	1.23
fop	7.6	2.56	0.90
h2	110.6	0.87	0.42
jython	86.9	2.02	1.42
luindex	13.7	6.48	0.67
lusearch	2.7	123.13	21.81
lusearch-fix	2.7	121.78	21.57
pmd	7.9	8.65	5.02
sunflow	14.2	49.70	6.27
tradebeans	82.8	2.97	1.90
tradesoap	36.0	13.71	6.40
xalan	3.5	60.34	12.14

6.2 基本的オーバヘッド

複製永続化には，オブジェクトを一切永続化しなくても発生する基本的オーバヘッドが存在する．その原因は以下の二つである．

1. 書込み中に他のスレッドがそのオブジェクトを永続化するのを防ぐためのロック
2. double update が必要かどうかのチェック

前者は，図 8 の `CAS` 命令によるもので，参照を書き込む `putfield` のたびに実行される．後者は，チェック自身は軽量だが，プリミティブ型の値を書き込む時には，チェックの前に `MFENCE` 命令が必要になり（図 9 参照），このコストは無視できない．

none-durable の設定を用いて，基本的オーバヘッドを計測した．図 10 の左の棒でオリジナルの HotSpot VM を基準にした実行時間の比と各書込みバリアに費した時間の内訳を示す．‘no barrier’ はバリアを無効化した時の実行時間，‘primitive write’ はプリミティブ型の書込みバリアだけを有効にした時の実行時間の差分，‘reference barrier’ は，さらに参照の書込み

バリアを有効にした時の実行時間の差分である．fop, luindex, luindex-fix, sunflow については，参照の書込みバリアを有効にた方が参照の書込みバリアを無効にするよりも高速に実行されたため，参照の書込みバリアの時間は 0 としている．同様に，h2, jython, xalan では，プリミティブ型の書込みバリアの時間を 0 としている．

全体のオーバヘッドは最大で avrora の 11.6 %だった．参照の書込みバリアとプリミティブ型の書込みバリアのどちらが大きなオーバヘッドになるかはベンチマークプログラムによって異なった．全体に，表 1 でプリミティブ型の書込みが頻繁なプログラムで，プリミティブ型の書込みバリアのオーバヘッドが大きい傾向にある．

書込みバリアのオーバヘッドの主な原因は，`MFENCE` 命令と `CAS` 命令と考えられる．これらの命令は CPU が out-of-order 実行できる命令に制限を付け，それがなくなると，CPU のストールを引き起こすためである．それを確認するために，`MFENCE` を消し，`CAS` を通常のロードとストア命令の組合わせに置き換えた実行を計測した．その結果を図 10 の右の棒で示す．この結果，`MFENCE` と `CAS` を実行しないようにした VM では，‘no barrier’ と同等の実行速度となった．このことから，`MFENCE` と `CAS` が主な原因と確認できた．

6.3 永続化のコスト

all-durable の設定を用いて，永続化のコストを計測した．表 2 にプログラムの実行を通して 1 秒当たりに永続化されたオブジェクトの数，ワード数を掲載する．また，永続化されたオブジェクトの 1 ワード当たりのキャッシュライン数 (cl/obj) と，1 秒当たりにフラッシュしたキャッシュラインの数，永続化オブジェクトへの書込みの頻度（pri と ref でそれぞれプリミティブ型と参照の書込みを表す）も掲載する．

図 11 に all-durable の実行時間とその内訳を示す．永続化のコストの大部分は double update(‘WUPD’) によるものだった．特に，表 2 で永続化オブジェクトへの書込みが多いプログラムで double update のオーバヘッドが大きい．luindex は書込みの頻度が lusearch などと比べると小さいが，このプログラムは並列度が小さいため，スレッド当たりの書込み頻度は大きい．なお，double update は NVM に書き込む時間と書込みをフラッシュする時間を含んでいる．NVM への書込みは，NVM を使って永続化するためには必ず行う必要がある．それ以外の永続化のコストは，NVM への書込みの時間と比べると十分に小さい．

6.4 AutoPersist との定性的な比較

複製永続化は，既存研究である AutoPersit で用いているリードバリアと，永続化オブジェクトから読み出す際の NVM へのアクセスを行わないように設計した．ここでは，書込みバリアのコストを定性的に比較する．

まず，永続化されていないオブジェクトへの書込みを考える．AutoPersist の書込みバリアでは，永続化されていないオブジェクトであっても，最低でも 1 回の `MFENCE` が必要になる．これは，書込みの最中にこのオブジェクトが永続化されたのを検出するためである．これに対して複製永続化では，プリミティブの値の書込みでは `MFENCE` が，参照の書込みでは `CAS` がそれぞれ 1 回必要になる．したがって，AutoPersist と複製永続化には，参照の書込みにおける `MFENCE` と `CAS` の違いがある．

次に永続化されたオブジェクトへの書込みを考える．永続化したオブジェクトへの書込みでは，複製永続化では DRAM と NVM の両方に書き込む必要がある．一方で，AutoPersist では NVM への書込みだけでよい．しかし，NVM への書込みは DRAM への書込みと比べて最大で 588 倍も遅い [33]．そのため，DRAM への書込みの時間は無視でき，AutoPersit と複製永続化のオーバヘッドはほぼ同等と言える．

7 関連研究

プログラミング言語の分野で NVM を利用する研究には，NVM を大容量な DRAM として用いる方向がある [30] [2]．この方向の研究では，永続化は興味の対象になっておらず，アプリケーションのスループットを高めたり，NVM の寿命への対処をしたり，消費電力を抑えることが興味の対象になっている．

もう一つの研究の方向はデータを永続化する方向で

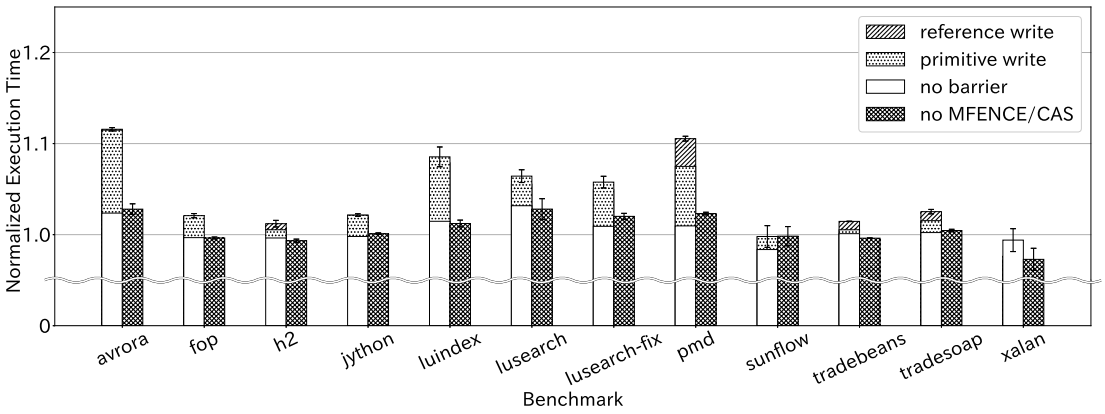


図 10　none-durable の設定での実行時間.

表 2　オブジェクトの永続化と永続化オブジェクトへの書込み回数.

	persistent objects				persistent writes		
name	(Kobj/s)	(Kword/s)	cl/obj	(Mcl/s)	pri (M/s)	ref (M/s)	total (M/s)
avrora	4.5	32.1	1.8	0.0	6.3	0.3	6.6
fop	9.9	145.4	2.7	0.0	0.2	0.0	0.2
h2	19.7	215.7	2.2	0.0	0.3	0.2	0.5
jython	106.8	1160.0	2.2	0.2	0.3	0.1	0.5
luindex	3.3	60.6	3.2	0.0	1.3	0.2	1.4
lusearch	203.6	1439.8	1.8	0.4	24.0	4.4	28.4
lusearch-fix	209.7	1483.1	1.8	0.4	24.7	4.5	29.2
pmd	86.9	992.7	2.3	0.2	0.0	0.0	0.1
sunflow	1.4	21.6	2.8	0.0	0.0	0.0	0.0
tradebeans	180.5	1667.9	2.0	0.4	0.2	0.2	0.5
tradesoap	415.0	5437.1	2.5	1.0	3.4	1.2	4.6
xalan	231.5	10657.0	6.6	1.5	17.7	3.2	20.9

ある．この方向では，どのように NVM をプログラミング言語に組み込むかが議論されている．Mnemosyne [29] と NV-Heaps [13] は NVM とトランザクションを組合わせた初期の研究である．これを発展させた，永続化トランザクションは盛んに研究されている [7] [11] [8] [16]．Atlas [12] は NVM とロックを組み合わせたシステムである．このプログラミングモデルは Boehm [10] によって議論されている．Espresso [32] は NVM をマネージドランタイムである Java VM に組合わせたシステムである．しかし，Espresso は NVM をトランザクションと組合わせているという点で，以前のシステムを踏襲している．

マネージドランタイムでは，トランザクションやロックよりも密接に言語に連携した永続化モデルが可能になる．マネージドランタイムでは，書込みバリアによって自動的に書き込んだデータのキャッシュラインをフラッシュできる．AutoPersist [24] はキャッシュラインのフラッシュにランタイムシステムが責任を持つシステムである．AutoPersist は，研究用の Java VM である Maxine VM [20] に到達可能性による永

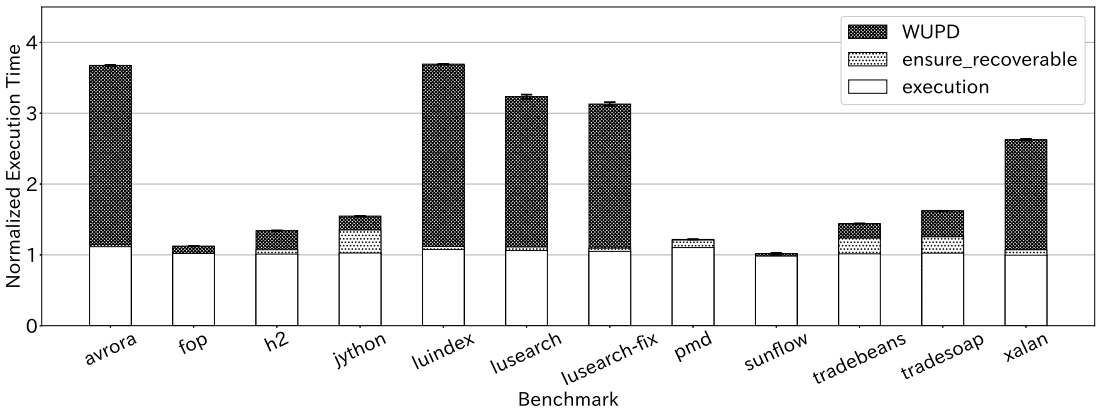


図 11　all-durable の設定での実行時間（オリジナル VM の時間で正規化）．

続化 [23] を実装した．AutoPersist は永続化するオブジェクトを直ちに NVM に移動させるアプローチをとった．このバリアのオーバヘッドは QuickCheck [25] の文献で評価されており，JIT を有効にした Maxine VM の実行の 8.8 %程度と報告されている．しかし，QuickCheck は AutoPersist の実現に必要なメモリバリア（`MFENCE`）のオーバヘッドを計測していない．そのため，8.8 %はメモリバリアの時間を含まないオーバヘッドと考えられる．P-INSPECT [19] はバリアのオーバヘッドを緩和するためにハードウェアを拡張することを提案している．GCPersist [31] はオブジェクトの永続化を次の GC まで遅延することで，オブジェクトの永続化がすぐに行われない代わりにオーバヘッドを大幅に削減した．

到達可能性に基く永続化のモデルは，orthogonally persistence [6] [5] [3] に関する一連の研究で提案されたものである．Atokinson ら [6] は一般的な二次記憶を使ってオブジェクトを永続化するシステムを提案した．PJama プラットフォーム [4] はその実装である．

複製永続化では，GC のテクニックを利用している．複製は Nettles [21] らによって提案された．double update バリアは複製に基づく GC [17] [26] [28] で使われている．さらに，複製永続化では Transactional Sapphire [28] の並行コピープロトコルを利用している．

8　おわりに

本研究では，オブジェクトの到達可能性に基づく永続化を実現する新しいアルゴリズムとして，複製永続化を提案した．このアルゴリズムは，既存研究である AutoPersist のオーバヘッドの原因であるリードバリアと，永続化されたオブジェクトからの読み出しに NVM へのアクセスを伴う点を解決している．また，書込みバリアで行うメモリバリアを伴う命令の回数は AutoPersist と同等である．

アルゴリズムの設計では，マルチスレッド環境で起こる可能性のある競合を指摘し，それを解消するアルゴリズムを示した．さらに，OpenJDK の HotSpot VM にプロトタイプを実装し，正しく動作することを確認した．

JIT コンパイラを有効にした HotSpot VM に実装し，先行研究との定量的な評価を行うことは今後の課題である．

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

参 考 文 献

[1] Adve, S. V. and Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial, *Computer*, Vol. 29, No. 12(1996), pp. 66–76.

[2] Akram, S., Sartor, J. B., McKinley, K. S., and Eeckhout, L.: Write-rationing garbage collection for hybrid memories, *Proceedings of the 39th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018, Foster, J. S. and Grossman, D.(eds.), ACM, 2018, pp. 62–77.

[3] Atkinson, M. P., Daynès, L., Jordan, M. J., Printezis, T., and Spence, S.: An Orthogonally Persistent Java, *SIGMOD Rec.*, Vol. 25, No. 4(1996), pp. 68–75.

[4] Atkinson, M. P. and Jordan, M. J.: Issues Raised by Three Years of Developing PJama: An Orthogonally Persistent Platform for Java, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, Beeri, C. and Buneman, P.(eds.), Lecture Notes in Computer Science, Vol. 1540, Springer, 1999, pp. 1–30.

[5] Atkinson, M. P., Jordan, M. J., Daynès, L., and Spence, S.: Design Issues for Persistent Java: A Type-Safe, Object-Oriented, Orthogonally Persistent System, *Proceedings of the 7th Workshop on Persistent Object Systems, Cape May, New Jersey, USA, 1996*, Connor, R. C. H. and Nettles, S.(eds.), Morgan Kaufmann, 1996, pp. 33–47.

[6] Atkinson, M. P. and Morrison, R.: Orthogonally Persistent Object Systems, *VLDB J.*, Vol. 4, No. 3(1995), pp. 319–401.

[7] Avni, H., Levy, E., and Mendelson, A.: Hardware Transactions in Nonvolatile Memory, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, Moses, Y.(ed.), Lecture Notes in Computer Science, Vol. 9363, Springer, 2015, pp. 617–630.

[8] Baldassin, A., Murari, R., de Carvalho, J. P. L., Araujo, G., Castro, D., Barreto, J., and Romano, P.: NV-PhTM: An Efficient Phase-Based Transactional System for Non-volatile Memory, *Euro-Par 2020: Parallel Processing - 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24-28, 2020, Proceedings*, Malawski, M. and Rzadca, K.(eds.), Lecture Notes in Computer Science, Vol. 12247, Springer, 2020, pp. 477–492.

[9] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, ACM Press, October 2006, pp. 169–190.

[10] Boehm, H. and Chakrabarti, D. R.: Persistence programming models for non-volatile memory, *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, Santa Barbara, CA, USA, June 14 - 14, 2016*, Flood, C. H. and Zhang, E. Z.(eds.), ACM, 2016, pp. 55–67.

[11] Castro, D., Romano, P., and Barreto, J. P.: Hardware Transactional Memory Meets Memory Persistency, *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*, IEEE Computer Society, 2018, pp. 368–377.

[12] Chakrabarti, D. R., Boehm, H., and Bhandari, K.: Atlas: leveraging locks for non-volatile memory consistency, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Black, A. P. and Millstein, T. D.(eds.), ACM, 2014, pp. 433–452.

[13] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S.: NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Gupta, R. and Mowry, T. C.(eds.), ACM, 2011, pp. 105–118.

[14] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M.: On-the-Fly Garbage Collection: An Exercise in Cooperation, *Commun. ACM*, Vol. 21, No. 11(1978), pp. 966–975.

[15] Friedman, M., Petrank, E., and Ramalhete, P.: Mirror: making lock-free data structures persistent, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Freund, S. N. and Yahav, E.(eds.), ACM, 2021, pp. 1218–1232.

[16] Genç, K., Bond, M. D., and Xu, G. H.: Crafty: efficient, HTM-compatible persistent transactions, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Donaldson, A. F. and Torlak, E.(eds.), ACM, 2020, pp. 59–74.

[17] Hudson, R. L. and Moss, J. E. B.: Sapphire: copying garbage collection without stopping the world, *Concurr. Comput. Pract. Exp.*, Vol. 15, No. 3-5(2003), pp. 223–261.

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

[19] Kokolis, A., Shull, T., Huang, J., and Torrellas, J.: P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks, *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, IEEE, 2020, pp. 509–524.

[20] Kotselidis, C., Clarkson, J., Rodchenko, A., Nisbet, A., Mawer, J., and Luján, M.: Heterogeneous Managed Runtime Systems: A Computer Vision Case Study, *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*, ACM, 2017, pp. 74–82.

[21] Nettles, S. and Jr., J. W. O.: Real-Time Replication Garbage Collection, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Cartwright, R.(ed.), ACM, 1993, pp. 217–226.

[22] Rudoff, A.: Persistent Memory Programming, *login Usenix Mag.*, Vol. 42, No. 2(2017).

[23] Shull, T., Huang, J., and Torrellas, J.: Defining a high-level programming model for emerging NVRAM technologies, *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*, Tilevich, E. and Mössenböck, H.(eds.), ACM, 2018, pp. 11:1–11:7.

[24] Shull, T., Huang, J., and Torrellas, J.: AutoPersist: an easy-to-use Java NVM framework based on reachability, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, McKinley, K. S. and Fisher, K.(eds.), ACM, 2019, pp. 316–332.

[25] Shull, T., Huang, J., and Torrellas, J.: QuickCheck: using speculation to reduce the overhead of checks in NVM frameworks, *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, Providence, RI, USA, April 14, 2019*, Sartor, J. B., Naik, M., and Rossbach, C.(eds.), ACM, 2019, pp. 137–151.

[26] Ugawa, T., Iwasaki, H., and Yuasa, T.: Improved replication-based incremental garbage collection for embedded systems, *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, Vitek, J. and Lea, D.(eds.), ACM, 2010, pp. 73–82.

[27] Ugawa, T. and Jones, R.: Model Checking Transactional Sapphire, March 2018. https://kar.kent.ac.uk/67197/.

[28] Ugawa, T., Ritson, C. G., and Jones, R. E.: Transactional Sapphire: Lessons in High-Performance, On-the-fly Garbage Collection, *ACM Trans. Program. Lang. Syst.*, Vol. 40, No. 4(2018), pp. 15:1–15:56.

[29] Volos, H., Tack, A. J., and Swift, M. M.: Mnemosyne: lightweight persistent memory, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Gupta, R. and Mowry, T. C.(eds.), ACM, 2011, pp. 91–104.

[30] Wang, C., Cao, T., Zigman, J., Lv, F., Zhang, Y., and Feng, X.: Efficient Management for Hybrid Memory in Managed Language Runtime, *Network and Parallel Computing - 13th IFIP WG 10.3 International Conference, NPC 2016, Xi'an, China, October 28-29, 2016, Proceedings*, Gao, G. R., Qian, D., Gao, X., Chapman, B. M., and Chen, W.(eds.), Lecture Notes in Computer Science, Vol. 9966, 2016, pp. 29–42.

[31] Wu, M., Chen, H., Zhu, H., Zang, B., and Guan, H.: GCPersist: an efficient GC-assisted lazy persistency framework for resilient Java applications on NVM, *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, Nagarakatte, S., Baumann, A., and Kasikci, B.(eds.), ACM, 2020, pp. 1–14.

[32] 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, Williamsburg, VA, USA, March 24-28, 2018*, Shen, X., Tuck, J., Bianchini, R., and Sarkar, V.(eds.), ACM, 2018, pp. 70–83.

[33] 松本康太郎, 高田喜朗, 鵜川始陽: 不揮発性メモリを用いた Java オブジェクト永続化のオーバヘッドの調査, コンピュータ ソフトウェア, Vol. 38, No. 2(2021), pp. 2_14–2_19.