プログラムを停止させない定期的な不揮発性メモリへのチェックポインティング

中田 昌輝　鵜川 始陽　佐藤 重幸

高い応答性が求められるサーバーアプリケーションでは，インメモリデータベースが広く利用されている．しかしインメモリデータベースは電源断が起きるとデータを消失してしまう．その解決策として不揮発性メモリへの定期的なチェックポインティングが有望である．既存研究の *CpNvm* は読み込みが多いワークロードにおいて高い性能を示す反面，チェックポインティングの際にユーザプログラムを停止させるため，応答性を低下させてしまう．そこで本研究は *CpNvm* に対し double buffering と Copy-on-Write を組み合わせ，チェックポインティングをユーザプログラムと並行に実行する手法を提案する．提案手法がユーザプログラムの高い応答性を維持させることと，読み込みが多いワークロードで性能が良いことを確認した．

1　はじめに

高い応答性を求められるサーバーアプリケーションは，インメモリデータベースが広く利用されている．インメモリデータベースは，全てのデータをメモリ（DRAM）上で管理するために，実行効率が良く，高い応答性を維持できる．その反面，持続性のあるストレージ（SSD や HDD）にデータを置かないために，電源断などの故障に対して脆弱である．

そこで有望なのは，持続性を持ちながら，メモリとして扱える不揮発性メモリ（NVM）である．NVM は，DRAM に比べてやや低速ではあるが，大容量であり，ストレージの代替利用が可能である．メモリ上のデータ構造を，適切に NVM 上に配置して管理すれば，アプリケーションの耐故障性と実行効率を両立できる見込みがある．しかし，単に更新データを NVM へ書き出すだけでは，故障からの回復時に，データ不整合が生じてしまう．意味的に不可分な一連のメモリ操作（トランザクション）に対して，故障時の不可分性（failure atomicity）を保証する永続化が必要になる．

この永続化をデータ構造に対して効率的に実現するには，buffered durable linearizability [12] に基づくアプローチが有望であることが知られている．これは，トランザクションにおけるデータ構造への書き込みをロギングして，適当な時間間隔（エポック）で，その書き込みログの内容を NVM 上のデータ構造に反映するものである．故障時に直近のエポックにおける書き込みが犠牲になる代わりに，NVM への書き込みの反映をアプリケーションと独立に処理できるようになり，永続化の処理全体が効率化される．

この buffered durable linearizability に基づく永続化について，既存の実装方針は，大きく二つに分けられる．一つは，データ構造全体と書き込みログを NVM に配置するもの（Montage [18] [6] 方式と呼ぶ）．もう一つは，データ構造の最新版を DRAM に，永続版を NVM に配置し，DRAM への書き込みログを NVM に配置するもの（*CpNvm* [2] [3] 方式と呼ぶ）．両者には一長一短がある．

Montage 方式では，アプリケーションの書き込み時に，Copy-on-Write（CoW）でエポック毎に新しいオブジェクトを生成し，書き込みログに記録する．このログの追記は，書き込みを行うスレッドを止めず

---

* Periodic Checkpointing to Non-Volatile Memory without suspending the program

Masaki Nakata, Tomoharu Ugawa, and Shigeyuki Sato, 東京大学, The University of Tokyo.

に処理することが可能である．また，アプリケーション側の並行アクセスが正しく排他制御されていれば（すなわち data-race-free ならば），書き込みログの反映は，アプリケーションと競合することはない．結果としてアプリケーションを停止させずに，永続化処理を並行させることができる．一方，最新版が NVM にあることから，読み込みは常に NVM から行うことになり，DRAM に比べて低速になる．

CpNvm 方式では，最新版が DRAM 上にあるため，アプリケーション側の読み込みにペナルティは生じず，通常の DRAM 上のデータ構造と同等の性能になる．また，アプリケーションの書き込み時は，書き込まれたアドレスを DRAM 上に記録するだけで，NVM に書き込むことはないので，オーバーヘッドが小さい．一方，書き込まれたデータをログに記録する際に，エポックを跨いだ最新版を参照すると，failure atomicity が壊れてしまう．したがって，書き込みログを追記する際は，アプリケーションを停止させる必要が生じる．結果として，アプリケーションの応答性が低下してしまう．

そこで本研究では，実行効率と応答性の両立を目指して，*CpNvm* 方式におけるアプリケーションと永続化処理の干渉を解消する実装手法を提案する．基本的な考え方は，Montage 方式における，1) CoW で NVM 上にコピーを生成すること，2) 書き込みログをエポック毎に区別して管理することを，*CpNvm* 方式に適応させるものである．具体的には，1) については，最新版が DRAM 上に置かれているべきなので，古いオブジェクトをコピーする方針を取る．2) については，書き込まれたアドレスを記録する DRAM 上のビットマップを，現在エポックと一つ前のエポックで区別して管理する方針を取る．これによって，アプリケーションが data-race-free である仮定の下で，アプリケーションを永続化処理によって止めることなく，*CpNvm* 方式の高い実行効率を享受できるようになる．そして，提案手法を利用したアプリケーションのスループットと応答性を実験的に評価した．その結果，Montage を上回るスループットと，*CpNvm* だけでなく Montage をも上回る応答性を確認した．

本研究の主な貢献は次の二つである．

- 高い実行効率と応答性を両立した，buffered durable linearizability に基づくデータ構造の永続化手法を開発した（4 章）．これは，*CpNvm* [2] [3] におけるアプリケーションと永続化処理が干渉する問題を，Montage [18] [6] の手法を *CpNvm* に適応させることで，解消したものである．
- 提案手法を利用したアプリケーションのスループットと応答性を実験的に評価した（5 章）．マイクロベンチマークと YCSB ベンチマークの両方について，提案手法は，スループットで Montage を上回り，応答性で *CpNvm* と Montage の両方を上回った．

2 Periodic Persistence

Periodic Persistence [16] とは buffred durable linearizability [12] に基づく手法である．アプリケーションの実行をエポックに分割し，故障後はデータ構造を直近のエポック終了時点の状態に復元する．永続化は failure atomicity を保証するために，定期的にアプリケーションを停止させたり，NVM へ保存するべきオブジェクト（永続化オブジェクトと呼ぶ）の書き込みはアトミック命令に限定したりする．

DRAM 上にデータ構造を構築する際，永続化において failure atomicity を保証するために，定期的にチェックポインティングを行う．チェックポインティングでは，エポックの切り替わりで，書き込みログを NVM 上にまとめて反映させる．このチェックポインティングは buffered durable linearizability が満たされるように，アプリケーションを停止させて実行する．

3 *CpNvm*

3.1 概要

CpNvm は DRAM 上にデータ構造を配置し，Periodic Persistence に基づく永続化を行う C++用ライブラリである．チェックポインティングは failure atomicity が満たされるように，アプリケーションを停止させてから行われる．読み込みの際は DRAM 上のデータ構造を直接参照するため，高速にアクセスす

```
1  void insert(K key, V val) {
2    std::lock_guard lk(List.lock);
3    ListNode* new_node = PNEW(ListNode, key, val);
4    mark(tail, sizeof(ListNode));
5    List->tail->next = new_node;
6    List->tail = new_node;
7    checkpoint();
8  }
```

図 1　永続化リストの insert 関数コード例

ることができる．

3.2　API

CpNvm はあらゆるデータ構造に対応できる永続化ライブラリであり，表 1 で示すような API を用いてデータの永続化を行う．*CpNvm* はリカバリーのための API も存在するが，本論文ではリカバリーについて言及しないため割愛する．

CpNvm ではトランザクションの最後やアプリケーションが停止可能なタイミングに，`checkpoint()` を記述することで，定期的にチェックポインティングを実行する．また，NVM へ保存するべきデータ構造に対しては，書き込み時に `mark(addr, size)` と記述する．*CpNvm* がチェックポインティングを行う際には，`checkpoint()` の位置でユーザスレッドの実行を止める．これらの API を用いて，永続化リストにおける `insert` 関数の実装例を図 1 に示す．

NVM へ保存するべきオブジェクトを作成する際は，必ず専用のマクロである `PNEW` を用いる（3 行目）．この `PNEW` マクロを用いることで NVM 上の複製を作成することを可能としている．また永続化オブジェクトの中身を書き換える際はその箇所に `mark` 関数を挿入する（4–5 行目）．最後に，この `insert` 関数は一つの transaction なので，トランザクションの終わりに `checkpoint` 関数を挿入する．

3.3　永続化の流れ

CpNvm の永続化の流れを図 2 に示す．また，表 1 の API や図 2 の処理の疑似コードを図 3 に示す．図 2 に示すように *CpNvm* は大きく分けて四つの要素で構成されている．DRAM 上にはデータ構造を格納する領域（DRAM ヒープと呼ぶ），書き込んだアドレスを記録するビットマップとアドレスリストがある．NVM 上にはチェックポインティングの際に作成する redo ログ，DRAM ヒープの複製である NVM ヒープがある．ビットマップとアドレスリストが記録するアドレスは DRAM ヒープを 64byte 単位で区切って管理する．ビットマップはグローバルに書き込んだアドレスを記録し，アドレスリストはスレッドごとに書き込んだアドレスを記録する．

CpNvm ではアプリケーションを実行するスレッド（ユーザスレッドと呼ぶ）と並行して永続化処理を担うスレッド（永続化スレッドと呼ぶ）が動作する．永続化スレッドは複数存在し，それぞれ複数のユーザスレッドと対応付けられて動いている．

2 種類のスレッドによる *CpNvm* の永続化処理は 3 種類によって構成され，書き込みバリアとチェックポインティングがある．書き込みバリアは図 2 中の①に示す**実行処理におけるマーキング**である．ユーザスレッドが書き込む際は `mark` 関数を記述し，書き込み先のアドレスを記録する．このマーキングはビットマップの対象のビットが立っていない場合のみである（図 3 の 13–15 行目）．これにより，記録するアドレス数を最小限に抑え，同じアドレスに対する書き込みの高速化が図れる．

チェックポインティングはまず，ユーザスレッドによる② **redo ログの作成**を行う．ユーザスレッドがスナップショットをとるべきと判断すると，この処理が始まる．ここで作成する redo ログは書き込み先のアドレスと書き込みデータの組を記録する．ユーザスレッドはグローバルに停止することが可能な状況下（トランザクション終了時など）で定期的に redo ログの作成処理を行う．この際，全てのユーザスレッドで同期を取り，アプリケーションを停止させてからこの処理が始めることで，整合性の取れた状態を得る．ビットマップやアドレスリストには直近のエポック内で書き込まれたアドレスしか記録されていない．そのため記録されたアドレスを元に redo ログに記録すべきデータを DRAM ヒープから得る．

redo ログの作成処理はユーザスレッドの `checkpoint` 関数を契機に行われる．この関数内の

`init(filename)`	永続化処理のための初期処理
`finalize()`	プログラム終了時の永続化完了処理
`thread_init()`	ユーザスレッドにおける永続化処理のための初期処理
`transaction_start()`	トランザクション開始時処理 (*CpNvm* では用いない)
`transaction_end()`	トランザクション終了時処理 (*CpNvm* では用いない)
`mark(addr, size)`	書き込みバリア
`checkpoint()`	チェックポイント処理 (提案手法では用いない)
`PNEW(type, ...)`	永続化オブジェクトの `new`
`PDELETE(type, ptr)`	永続化オブジェクトの `delete`

表 1　永続化処理のための API

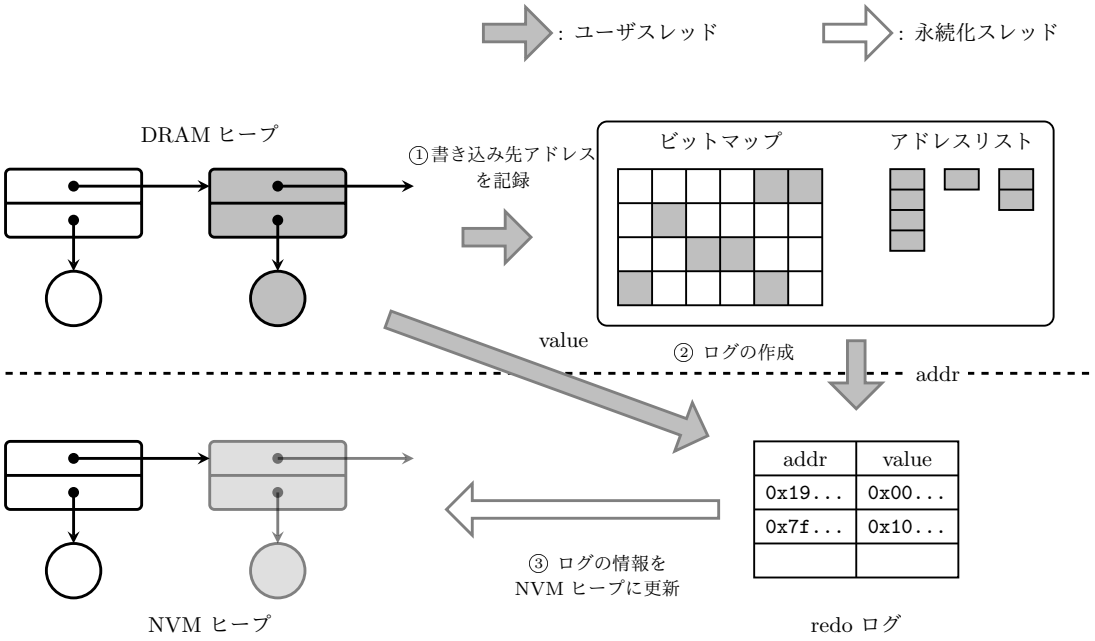


図 2　*CpNvm* の概要図

redo ログの作成では書き込み先のアドレスは主にアドレスリストを参照して得る．アドレスリストを参照することで，ビットマップ全部を走査する必要がない，redo ログ作成の並行実行が可能，など高速化に寄与する．さらに，redo ログの作成の度にビットマップの対象となるビットを下ろす．すると，アドレスリストとビットマップが記録するアドレス情報は同じため，この処理が終わるとビットマップはクリアされている．したがって，ビットマップは各エポックで同一のものが用いることができる．

チェックポインティングの二つ目の処理は永続化スレッドによる③ **バックグラウンドで redo ログの情報を NVM ヒープに更新すること**である．ユーザスレッドは redo ログの作成終了後，アプリケーションの実行を再開する．それと同時に永続化スレッドが起動し，ユーザスレッドの実行と並行して（①と並行して），redo ログの情報を NVM ヒープにデータを更新する（図 3 の `persist` 関数）．

以上の永続化の流れで，NVM ヒープと redo ログの情報から整合性の取れるデータ構造を復元するこ

```
1  time_t last_checkpoint_time;
2  bool bitmap[HEAP_SIZE/sizeof(CacheLine)];
3
4  struct ThreadContext {
5    list<intptr_t> addr_list;
6    list<pair<intptr_t,CacheLine>> log; // on NVM
7  };
8  ThreadContext ctx[NUM_THREADS]; // indexed by thread
     ID
9
10 void mark(intptr_t addr, size_t size) {
11   auto id = my_thread_id();
12   for (int i = 0; i < size; i += sizeof(CacheLine))
13     if (!bitmap[(addr + i)/sizeof(CacheLine)]) {
14       ctx[id].addr_list.push_back(addr + i);
15       bitmap[(addr + i)/sizeof(CacheLine)] = true;
16     }
17 }
18
19 template <typename T, typename... Args>
20 T *pnew(Args... args) {
21   auto ptr = new T(args...);
22   mark(static_cast<intptr_t>(ptr), sizeof(T));
23   return addr;
24 }
25 #define PNEW(type, ...) pnew<type>(__VA_ARGS__)
26
27 template <typename T>
28 void pdelete(T* ptr) {
29   mark(static_cast<intptr_t>(ptr), sizeof(T));
30   delete obj;
31 }
32 #define PDELETE(type, ptr) pdelete<type>(ptr)
33
34 void checkpoint() {
35   if (current_time() - last_checkpoint_time <=
       INTERVAL)
36     return;
37
38   thread_barrier();
39   auto id = my_thread_id();
40   for (auto addr : ctx[id].addr_list) {
41     bitmap[addr/sizeof(CacheLine)] = false;
42     auto value = *static_cast<CacheLine*>(addr);
43     ctx[id].log.push_back({addr, value});
44   }
45   ctx[id].addr_list.clear();
46   thread_barrier();
47   sfence();
48   renew_last_checkpoint_time();
49 }
50
51 void persist() {
52   for (int i = 0; i < NUM_THREADS; i++) {
53     for (auto& [addr, value] : ctx[i].log) {
54       auto nvm_addr = translate_to_nvm(addr);
55       memcpy(nvm_addr, &value, sizeof(CacheLine));
56       clwb(nvm_addr);
57     }
58     ctx[i].log.clear();
59   }
60   sfence();
61 }
```

図 3　*CpNvm* の疑似コード

とが可能となる．この整合性の取れるデータ構造は直近のエポック終了時点の状態であり，復元方法は 2 パターン存在する．一つ目は NVM ヒープに整合性の取れた直近のエポック終了時点の状態が保存されている場合である．永続化スレッドが永続化処理待機中（①, ②の処理中）では NVM ヒープ内に一つ前のエポック終了時点のデータ構造が保存されている．

二つ目は NVM ヒープと redo ログに整合性の取れた状態が保存されている場合である．永続化スレッドが永続化処理を行っている最中（③の処理中）では NVM ヒープ内に二つ前のエポック終了時点のデータ構造に redo ログのデータが一部反映された状態が保存されている．そのため，この時点で故障が起きた際は redo ログから NVM ヒープにデータの更新作業を行うことでクラッシュ直前のエポック終了時点の状態を復元可能である．

3.4　永続化スレッドの並列化

永続化スレッドの `persist` 関数における NVM ヒープへの更新処理は並列実行が可能である（図 3 の 51–56 行目）．*CpNvm* が作成する redo ログには，エポックの終了時点のデータのみ保存されている．そのため，各ユーザスレッドが作成するログには順序関係が存在しない．したがって，どの順番でログを NVM ヒープに反映させても，最終的には整合性の取れるデータが NVM ヒープ上に永続化される．このため，*CpNvm* の著者は永続化スレッドを二つ用意して実装している．

3.5 *CpNvm* の問題点

3.1 節で述べたように failure atomicity を満たすために，*CpNvm* はアプリケーションを停止させてチェックポイントを取る．この停止は図 2 における②の redo ログの作成が完了するまで行われる．したがって，書き込みの割合が大きいワークロードの場合，redo ログの作成に時間がかかり停止時間が長くなる．しかし，インメモリデータベースでは高い応答性が要求されるため，*CpNvm* のチェックポインティングによる停止はインメモリデータベースに適さないという問題がある．

4 提案手法

4.1 方針

提案手法は，3.5 節の問題を解決し，アプリケーションを停止させないようにチェックポインティングする手法である．そのために提案手法では *CpNvm* を次の方針で変更する．

1. *CpNvm* ではユーザスレッドが行っていた redo ログの作成も永続化スレッドが実行する．
2. 二つの隣接するエポックのトランザクションが同時に実行することを許すが，チェックポインティングではエポック境界のスナップショットをとる．

4.2 スライディングビュー

トランザクションの途中のメモリ状態を永続化しないようにするために，チェックポインティングはトランザクションが実行していないときに行う必要がある．しかし，ユーザスレッドが複数あると，どのスレッドもトランザクションを実行していない瞬間が定期的に訪れるとは限らない．

そこで，各スレッドのトランザクション完了時のビューを結合した**スライディングビュー**[4] のスナップショットをとる．図 4 のようにスナップショットを開始する時点で，いくつかのユーザスレッドがトランザクションを実行中である場合，それらのトランザクションに影響されない部分のメモリは，スナップショット開始時の状態を保存する．一方，実行中のトランザクションによって書き込まれるメモリは，トランザクション完了時のメモリ状態を保存する．これにより，図 4 の点線のような，スレッド毎にタイミングがずれたスナップショットがとられることになる．

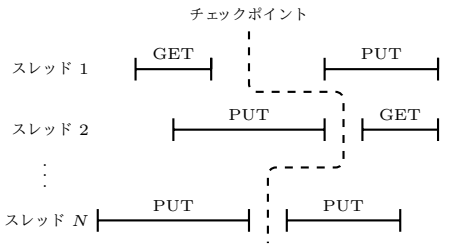


図 4　スライディングビューに基づくスナップショット

これを実現するために，提案手法では，チェックポインティング開始時に実行中のトランザクションの完了を待ってからスナップショットをとり始める．しかし，待っている間に次のエポックのトランザクションを実行することを許すので，スナップショットをとるのに必要なデータが書き換えられる可能性がある．そこで，double buffering（4.6 節）と CoW（4.7 節）を使ってこれを防ぐ．

4.3 Data-race-freedom の仮定

チェックポインティング時に実行中のトランザクション X がアクセスする範囲に，次のエポックのトランザクション Y もアクセスすると，トランザクション X 完了時のビューに新しいエポックのトランザクション Y の結果が混ざってしまう．その結果，他のスレッドのビューと統合したスライディングビューに新しいエポックの実行結果が混ざってしまい，buffered durable linearizability を満たさなくなる．

本研究では，このようなことを避けるために，ユーザスレッドに data-race-freedom を要求する．より精確には，次の制約を課す．

> 同時に実行される二つのトランザクションが同じメモリアドレスをアクセスするときは，どちらのトランザクションもそのメモリアドレスに書き込んではならない．

4.4 API

3.2 節で説明した表 1 のとおりであり，*CpNvm* はトランザクション終了時に `checkpoint()`

を書き込んだのに対し，トランザクション前後で `transaction_start()`, `transaction_end()` と記述する．図 1 では，関数の開始時と終了時にこれらの API を呼び出すことで実行するトランザクションのエポックの管理を行う．

4.5 エポック番号

提案手法では，隣接する二つのエポックのトランザクションが同時に実行される．そこで，トランザクションの属するエポックを区別するために，エポック番号を明示的に管理する．各トランザクションにエポック番号を付与し，どのエポック内で実行しているかが分かるようになる．これらの変更を加えた提案手法の疑似コードを図 5 に示す．

エポック番号はグローバルなエポック番号とユーザスレッドが個々に保持するエポック番号の 2 種類用意する．ユーザスレッドが保持するエポック番号は現在実行しているトランザクションのエポック番号である．エポック番号が 0 のときはトランザクションを実行していないことを表す．ユーザスレッドはトランザクションを始める時にグローバルなエポック番号をスレッドのエポック番号にコピーする（図 5 の 14 行目）．トランザクション終了時にはスレッドのエポック番号を 0 に戻す（図 5 の 20 行目）．

一方で，グローバルなエポック番号の管理は永続化スレッドが担う．永続化スレッドはチェックポインティングを開始する際はこのエポック番号をインクリメントする（図 5 の 42 行目）．グローバルなエポック番号をインクリメントした後も，トランザクションを実行中のスレッドのエポック番号は変わらない．永続化スレッドは古いエポックのトランザクションの終了を待ってから（図 5 の 46–47 行目）チェックポインティングを始める．

4.6 Double Buffering

CpNvm では，ユーザスレッドが一つのビットマップのみを用いて書き込みバリアを行っていた．これは次のエポックになると，一つ前のエポックにおける redo ログの作成は完了しているからである．しかし，提案手法では次のエポックのトランザクションが一つ前のエポックにおける redo ログを作成中に，ビットマップを書き換える可能性がある．この問題はエポックごとにビットマップを用意することで解決する．

図 6 に示すように，次のエポックでのビットマップはアドレスの記録用，一つ前のエポックでのビットマップは redo ログ作成用，とビットマップに 2 種類の役割をもたせた（double buffering と呼ぶ）．

永続化対象のエポックは現在実行しているエポックの一つ前のエポックである．次のエポックでのトランザクションは一つ前のエポックのビットマップに書き込むことはない．そのため，永続化対象のビットマップは一つ前のエポックで書き込まれたアドレスの情報のみを持つ．それゆえ，一つ前のエポック終了時のスライディングビューを得るために必要なアドレス情報を正確に得ることが可能となる．

このビットマップはエポックごとに用意する必要はなく，二つのビットマップで十分である．3.3 節で述べたようにチェックポインティング終了後はビットマップは全てクリアされている．そのため，次のエポックでのビットマップは二つ前のビットマップを再利用することが可能である．したがって，エポックが切り替わる度に二つのビットマップの役割を変えて図 6 の処理を行う．

4.7 Copy on Write

CpNvm では，ユーザスレッドは書き込んだアドレスしか記録しない．書き込んだ値は DRAM ヒープから読み出せるからである．しかし，提案手法では次のエポックのトランザクションによって DRAM ヒープが書き換えられる可能性がある．この問題は，新しいエポックのユーザスレッドが書き込み時に redo ログの作成を手伝うことで解決する．

図 7 において，*CpNvm* と同じ処理を行った場合，書き込みデータにオブジェクトの参照を切り替えた後に永続化処理が行われると不整合が発生する．オブジェクトが保持するポインタは書き込みデータ（灰色の丸）を指すことになるが，書き込みデータは永続化されないためダングリングポインタとなる．

この解決のために，永続化が完了してない場合は上書きの前にデータの redo ログを作成 (copy) して

```
unsigned long global_epoch;
time_t last_checkpoint_time;
bool bitmap[2][HEAP_SIZE/sizeof(CacheLine)];

struct ThreadContext {
  unsigned long epoch;
  list<intptr_t> addr_list[2];
  list<pair<intptr_t,CacheLine>> log; // on NVM
};
ThreadContext ctx[NUM_THREADS]; // indexed by thread ID

void transaction_start() {
  auto id = my_thread_id();
  ctx[id].epoch = global_epoch;
}

void transaction_end() {
  auto id = my_thread_id();
  ctx[id].epoch = 0;
}

void mark(intptr_t addr, size_t size) {
  auto id = my_thread_id();
  auto cur = ctx[id].epoch % 2;
  auto old = (ctx[id].epoch - 1) % 2;
  for (int i = 0; i < size; i += sizeof(CacheLine))
    if (!bitmap[cur][(addr + i)/sizeof(CacheLine)]) {
      if (bitmap[old][(addr + i)/sizeof(CacheLine)]) {
        auto value_old = *static_cast<CacheLine*>(addr + i);
        ctx[id].log.push_back({addr + i, value_old});
      }
      bitmap[cur][(addr + i)/sizeof(CacheLine)] = true;
      ctx[id].addr_list[cur].push_back(addr + i);
    }
}

void persist() {
  if (current_time() - last_checkpoint_time <= INTERVAL)
    return;

  global_epoch++;

  // Corresponds to checkpoint() in CpNvm
  list<pair<intptr_t,CacheLine>> log; // on NVM
  for (int i = 0; i < NUM_THREADS; i++)
    wait_until([i](){ return ctx[i].epoch != global_epoch - 1; });
  for (int i = 0; i < NUM_THREADS; i++) {
    auto old = (global_epoch - 1) % 2;
    for (auto addr : ctx[i].addr_list[old]) {
      auto value = *static_cast<CacheLine*>(addr);
      log.push_back({addr, value});
      bitmap[old][addr/sizeof(CacheLine)] = false;
    }
  }
  ctx[id].addr_list[old].clear();
  for (int i = 0; i < NUM_THREADS; i++)
    wait_until([i](){ return ctx[i].epoch == 0; });
  sfence();

  // Corresponds to persist() in CpNvm
  for (auto& [addr, value] : log) {
    auto nvm_addr = translate_to_nvm(addr);
    memcpy(nvm_addr, &value, sizeof(CacheLine));
    clwb(nvm_addr);
  }
  for (int i = 0; i < NUM_THREADS; i++) {
    for (auto& [addr, value] : ctx[i].log) {
      auto nvm_addr = translate_to_nvm(addr);
      memcpy(nvm_addr, &value, sizeof(CacheLine));
      clwb(nvm_addr);
    }
    ctx[i].log.clear();
  }
  sfence();
  renew_last_checkpoint_time();
}
```

図 5　提案手法の疑似コード

から，書き込み (write) を行う．これにより，仮に永続化スレッドが新しい書き込みデータを参照するオブジェクトを永続化してしまったとしても，ユーザスレッドが作成した redo ログには整合性の取れている一つ前のエポックでのデータが記録されている．

書き込み先のデータが永続化が完了しているかどうかは，一つ前のエポックのビットマップを参照することで判別可能である．永続化が完了していないということは一つ前のエポックで書き込みが行われている．そのため，一つ前のエポックのビットマップを参照した際に，対象のビットが立っていれば永続化が完了していないことを判定できる．

また，現在のエポックのビットマップを参照した際に対象のビットが立っていれば，そのデータに対する

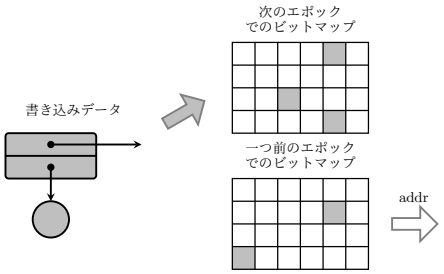


図 6　Double Buffering

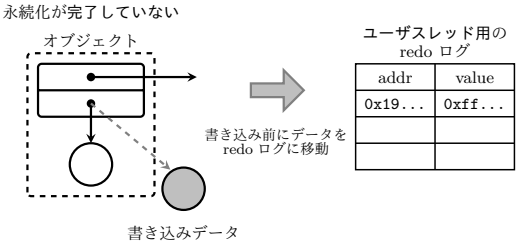


図 7　Copy on Write

redo ログは必ず作成されている．そのため，CoW の処理を行うか判断するのは，新しいエポックにおいて初めて対象のデータにアクセスしたときだけである（図 5 の 27–35 行目）．

NVM ヒープを更新する際は永続化スレッドが作成した redo ログとユーザスレッドが作成した redo ログに同じアドレスのデータが記録されている場合がある．その際はユーザスレッドが作成する redo ログを採用する．これは 4.7 節で述べた，永続化スレッドが作成する redo ログは不整合なデータが記録されている可能性があるからである．一方でユーザスレッドが作成する redo ログは必ず一つ前のエポックの整合性の取れたデータを記録する．したがって，永続化スレッドは永続化スレッドが作成した redo ログを先に NVM ヒープに反映させてから，ユーザスレッドが作成した redo ログを反映させる（図 5 の 61–77 行目）．これにより，永続化スレッドが作成した不整合なデータが含まれている redo ログによる更新を消すことが出来る．

4.8　NVM ヒープへの更新

永続化スレッドが redo ログを作成する処理が終わったあと，*CpNvm* と同様に NVM ヒープの更新処理を始める．ここで，*CpNvm* の `checkpoint()` の最後に行われるバリア同期と同様の処理が提案手法でも必要になる．それはユーザスレッドがまだ redo ログを作成している可能性があるためである．したがって，NVM ヒープの更新はユーザスレッドのトランザクションが終了してから行われる（図 5 の 56–58 行目）．

4.9　永続化スレッドの並列化

CpNvm と同様，永続化スレッドは並列化して処理を行うことができる．提案手法の `persist` 関数では，redo ログの作成処理および redo ログのデータを NVM ヒープへ更新する処理が並列化できる．redo ログの作成処理に関しては *CpNvm* の `checkpoint` 関数と同様の処理で並列化出来る．

NVM ヒープへ更新する際は 4.8 節で述べた永続化スレッドとユーザスレッドが作成した redo ログの反映順序に気をつけなければならない．そのため，永続化スレッドが作成した redo ログを反映したのち，永続化スレッド同士で同期を取ってからユーザスレッドが作成した redo ログの反映処理を行う．

永続化スレッドは，*CpNvm* では 2 スレッドを用いていたが，提案手法ではユーザスレッドと同数のスレッドを用いる．これは redo ログ作成を，少ないスレッド数で処理すると，ユーザスレッドの CoW が多く発生することを防ぐためである．

5　評価

本章では，既存研究と比較して実験を行い，次の 2 点を実験的に確認した．

1. 4.6, 4.7 節で説明した double buffering と CoW によって，プログラムを停止させないことを実現できているか（5.2 節）
2. 提案手法は実際のワークロードにおいても，*CpNvm* の強みである読み込み性能が維持されているか（5.3 節）

5.1 実験設定

実験環境は以下のとおりである．

- CPU: Intel Xeon Gold 6354(3.00GHz)
  - 18 コア 36 スレッド
- DRAM: DDR4 3200MHz 192GB
- NVM: Intel Optane DC Persistence Memory 256GB
- C++ コンパイラ: g++ 9.3.0
- OS: Ubuntu 20.04.3LTS

CPU はハイパースレッディングをオンにしており，turbo boost を切って 3.00GHz に固定している．また NUMA による NVM へのアクセス時のオーバーヘッドを避けるために，CPU は使用する NVM に直接つながっているもののみを使用している．提案手法はユーザスレッドと永続化スレッドが同じキャッシュを共有するように CPU を固定して計測を行っている．

また，比較実験に用いた既存研究は以下のとおりである．

- *CpNvm*: 3 章で説明した手法を我々が実装したものである．永続化スレッドの個数は著者らの実装に合わせて 2 個とした．
- Montage [18] [6]: NVM 上にコンテナデータ構造を構築し，コンテナとその要素を永続化する．wait free なデータ構造に対しても永続化が適用可能な手法である．実装は著者らが公開しているもの †1 を使用した．

実験結果において，データを永続化しないプログラムを DRAM と表記し，提案手法のライブラリを用いたデータ構造を Proposed と表記する．

メモリアロケータについて，先行研究の *CpNvm* [2] [3] は InCLL [8] で用いられるフリーリストによるアロケータを用いていたが，我々の *CpNvm* の実装では jemalloc†2 を使用している．また提案手法のアロケータも jemalloc がベースとなっているが，CoW 時の redo ログ作成における data-race を避けるために 64byte にアライメントするように変更を加えた．その他のプログラムについては，DRAM は jemalloc, Montage に関しては ralloc [5] を用いている．

実験で用いたベンチマークは主に二つである．一つ目は既存研究 Montage で用いられた Hashmap へのアクセスを行うベンチマークである．get, insert, remove をクエリとして各クエリでは Hashmap のバケットごとにロックを保持していることで data-race-free なプログラムである．また，get, insert, remove の割合を変えて実験することができ，それを g:i:r=0:1:1 のように書く．Hashmap のバケットの数は 1,000,000，書き込みする値のサイズは 1KB である．また定期的にチェックポインティング（や永続化処理）を行う手法に関しては，その間隔を 100ms とした．

二つ目は YCSB ベンチマーク [9] であり，memcached [1] 上に永続化ライブラリを挿入し，100ms おきにチェックポインティングを行う．メモリアクセスは zipfian 分布に従うワークロードを使用した．

5.2 応答性

どの程度停止しているかを各時間におけるクエリの数を計測することで確認した．データを永続化しないプログラムの実行 (DRAM の実行) における平均実行クエリ数の 10%, 30%, 50%を最低保証性能とする．この最低保証性能を下回った実行クエリ数の場合，その実行は停止と判断する．また，実行時間に対する停止時間の割合を棒グラフで表現する．さらに 1 回あたりに停止した時間を 0.5ms 未満，1ms 未満，5ms 未満，10ms 未満，50ms 未満，50ms 以上で区切って表示し，その割合によって評価を行う．

ユーザスレッド数は 1, 4, 16 スレッドで行った．

5.2.1 HashMap へのアクセスにおける応答性

まず，書き込み中心なワークロード (g:i:r=0:1:1) における実行で，100μs ごとに実行されたクエリ数を出し，どのようにチェックポインティングが行われているかを 1, 16 スレッドの結果をそれぞれ図 8, 9 に示す．

図 8, 9 でチェックポインティングが行われた位置に $\star$ 印を示す．*CpNvm* と提案手法のスナップショットは定期的に取られている．*CpNvm* は実行したクエリ数が 0 となる時間が数ミリ秒が存在している．一方で提案手法は 1 スレッドのときに，スナップショットを

†1 https://github.com/urcs-sync/Montage
†2 https://github.com/jemalloc/jemalloc

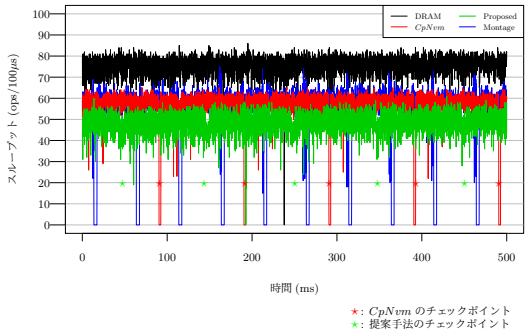


図 8　Hashmap へのアクセスにおける応答性（1 スレッド）

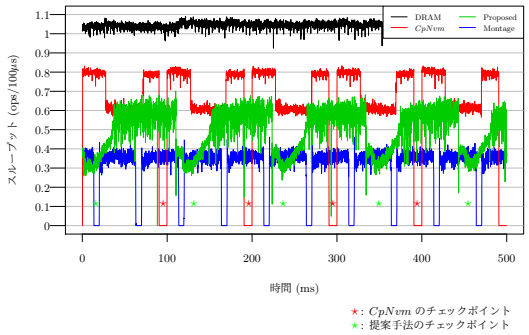


図 9　Hashmap へのアクセスにおける応答性（16 スレッド）

取るタイミングがほとんど判別付かないほど，安定したスループットである．16 スレッドの場合，チェックポインティングによりスループットが大きく低下した状態でも，Montage のスループットと同程度である．

図 9 において *CpNvm* のスループットが 0 ops/100ms, 600ops/100ms, 800 ops/100ms の 3 段階存在している．それぞれ redo ログの作成フェーズ，バックグラウンドで NVM ヒープの更新フェーズ，書き込みバリアのみ行っているフェーズである．

Montage に関してはチェックポイントの間隔の 2 倍の頻度で停止しているが，その原因は今後調査する予定である．

書き込み中心なワークロード (g:i:r=0:1:1) における停止割合の結果を図 10，読み込み中心なワークロード (g:i:r=18:1:1) における停止割合の結果を図 11 に示す．書き込み中心の場合，16 スレッド実行結果を示す図 10-(c) において，最低保証性能が DRAM の性能の 50%の際，提案手法と Montage が見切れている．提案手法では 40%近くが停止とみなされ，Montage については全ての実行が停止とみなされている．同様に読み込み中心の場合である図 11-(c) でも Montage が見切れており，78%が停止とみなされている．

提案手法はスレッド数が多くなると停止の割合が増える．書き込みメインのワークロードの結果を示した図 10 において，最低保証性能が DRAM の性能の 50%の場合，スレッド数が増えると *CpNvm* に比べ，停止時間が増える．これはユーザスレッドの 2 倍が CPU のコア数を超えるような実験では，永続化スレッドに CPU をとられてユーザスレッドの速度が低下しているためである．

5.2.2　YCSB ベンチマークにおける応答性

読み書きの割合が 1 : 1 であるワークロードの YCSB-A における応答性に関する結果を図 12 に示す．

ユーザスレッド数が 1 スレッド時の停止割合の結果である図 12-(a) において，Montage の性能が悪いため，最低保証性能が DRAM の性能の 50%の際は，38%停止している．

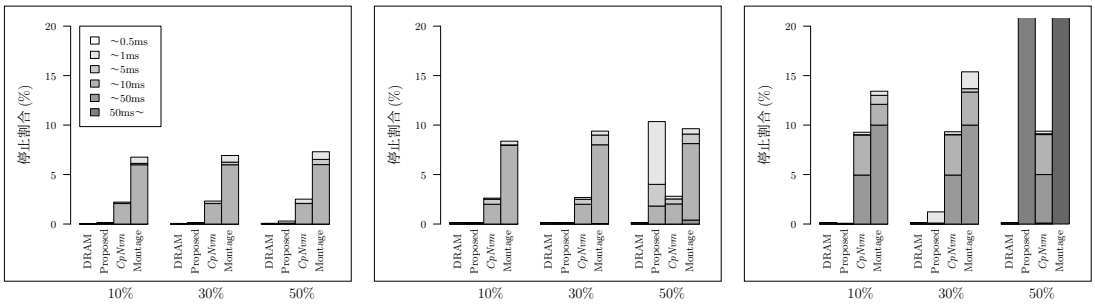
提案手法に関して，図 12-(c) に示す，16 スレッドで最大保証性能が DRAM の性能の 50%である場合，最も停止割合が多く，その割合は 0.31%である．しかし *CpNvm* は最大の場合に 6.5%停止し，Montage は 38%停止していることを考慮すると，停止時間はわずかであるといえる．さらに提案手法の場合は，このときの約 67%が，0.5ms 未満の停止である．

CpNvm と Montage はスレッド数や最低保証性能によらず，一定時間停止している．最低でも *CpNvm* は 3.4%停止，Montage は 7.5%停止している．さらにその停止は 5ms から 10ms を要するものである．

5.3　スループット評価

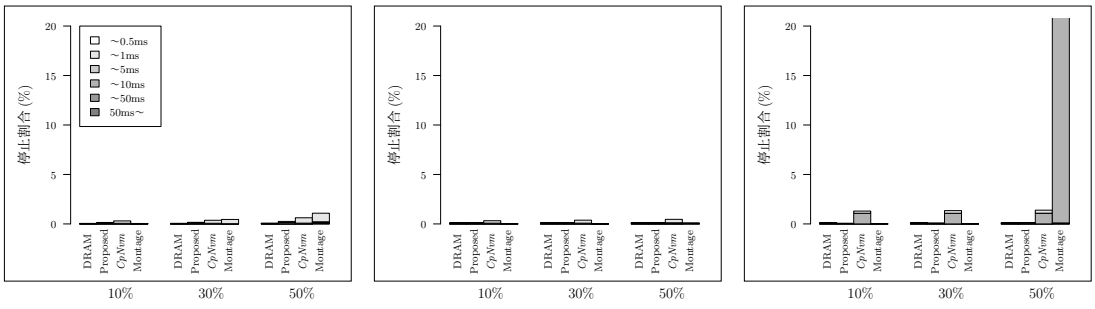
5.3.1　Hashmap

Hashmap へのアクセスを行うベンチマークでは書き込みメインなワークロード（g:i:r=0:1:1) と読み込



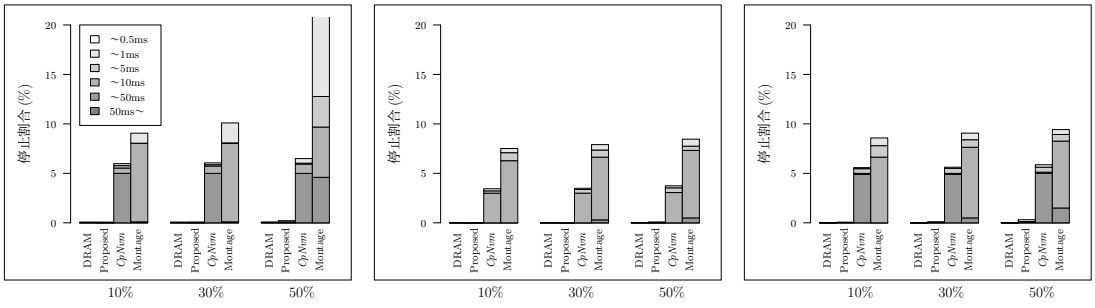
(a) 1 スレッド　(b) 4 スレッド　(c) 16 スレッド

図 10　Hashmap へのアクセス (g:i:r=0:1:1) における停止時間の割合



(a) 1 スレッド　(b) 4 スレッド　(c) 16 スレッド

図 11　Hashmap へのアクセス (g:i:r=18:1:1) における停止時間の割合

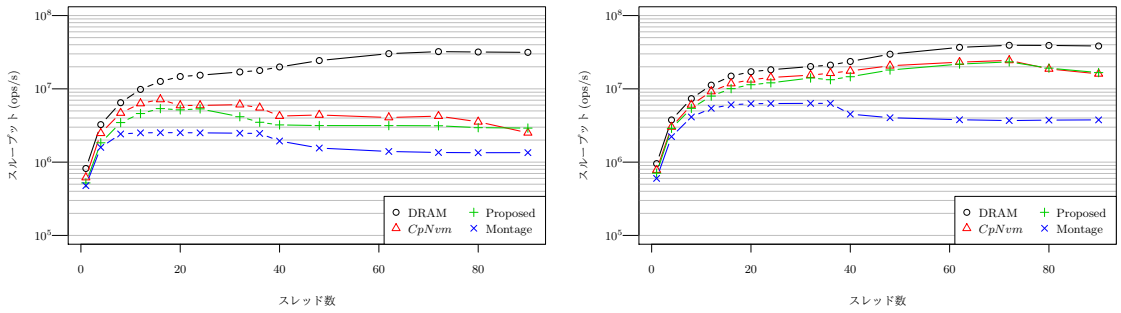


(a) 1 スレッド　(b) 4 スレッド　(c) 16 スレッド

図 12　YCSB ベンチマークへのアクセスにおける停止時間の割合

みメインなワークロード (g:i:r=18:1:1) の測定を行った．結果を図 13 に示す．

図 13-(a) に示す書き込み中心のワークロードでは，*CpNvm* が最も良い性能であることが分かる．*CpNvm* は 16 スレッドと 36 スレッドの間を境にスループットが大きく落ちている．これはユーザスレッドは 18 スレッドで CPU を全て使うこと，36 スレッドでハイパースレッディングを活用して CPU を全て使うこと



(a) 書き込み中心なワークロード（g:i:r=0:1:1）　(b) 読み込み中心なワークロード (g:i:r=0:1:1)

図 13　Hashmap への読み書きにおけるスループット

に起因している．また 90 スレッド実行のスループットは 80 スレッドの際と比べてスループットが大きく落ちる原因には，スレッド同期に時間がかかってしまっていることが挙げられる．

一方で提案手法は 16 スレッドと 20 スレッドの間を境にややスループットが落ちる結果になっている．これは永続化スレッドがユーザスレッドと同じ数だけ作られており，ユーザスレッドを 16 スレッド以上使う場合，18 個の CPU コアを全て利用しても 1 コアに 2 スレッド以上割り当てられるためである．スレッド数が増えてもスループットがあまり落ちないのは，*CpNvm* と異なりユーザスレッド同士の同期をとらないためである．

Montage については 36 スレッドと 40 スレッドの間を境にややスループットが落ちる結果になっている．提案手法は Montage よりもスループットがよく，最大で 2.2 倍スループットがよい．

図 13-(b) に示す読み込み中心のワークロードでは DRAM 上にデータを置く *CpNvm*, 提案手法が高い性能を示している．提案手法と *CpNvm* は DRAM と比べて最低でもそれぞれ 0.43 倍，0.42 倍と読み込み性能が高いことが確認できる．提案手法は *CpNvm* に比べ，最低でも 0.83 倍のスループットであるため，Montage と比較して *CpNvm* の強みを維持できているいえる．

5.3.2　YCSB ベンチマーク

読み書きの割合が 1 : 1 である YCSB-A と読み書きの割合が 19 : 1 である YCSB-B の二つのワークロードを用いて測定を行った．測定結果を図 14 に示す．

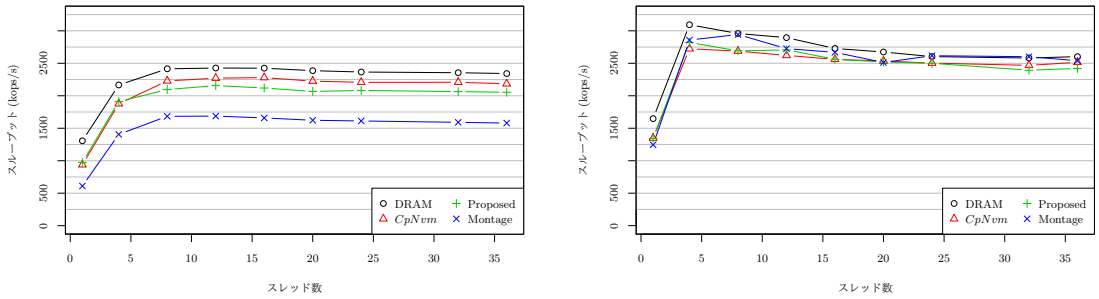
図 14-(a) に示す YCSB-A におけるスループットでは，*CpNvm* と提案手法は高いスループットである．DRAM と比較してそれぞれ最低でも 0.94 倍，0.89 倍のスループットであり，実際のアプリケーションでも高い性能を引き出せることがいえる．Montage は最低の場合，0.68 倍のスループットとなってしまう．これは Montage がたとえ一つのフィールド変数を書き換えても，エポックが変わってしまうとオブジェクト全体をコピーし直す必要があるためである．

図 14-(b) に示す YCSB-B におけるスループットでは，全体的に高い性能を示している．*CpNvm*，提案手法，Montage それぞれ最低でも 0.85 倍，0.89 倍，0.86 倍のスループットである．これは，どの手法も読み込み時は永続化のための特別な処理を行っていないことから高いスループットになるといえる．

6　関連研究

NVM を用いたデータやオブジェクトの永続化に関する初期の仕事では，NV-Heaps [7] や Mnemosyne [17] などの，トランザクション単位の永続化が主流であった．永続性トランザクションを，計算の実行・redo ログへの記録・NVM 上のデータの更新の 3 ステップに分離させる DudeTM [14] の技法は，*CpNvm* の原型となっている．

NVM 上に並行データ構造として，lock-free キュー [11] や lock-free Hashmap の SOFT [21] が開発されてきた．これらは個々のデータ構造を，効率的



(a) YCSB-A　　(b) YCSB-B

図 14　YCSB ベンチマークにおけるスループット

に NVM 上で永続化することが主眼である．NVTraverse [10] はこれらを一般化し，DRAM 上で動作するリンク構造のデータ構造の走査コードを，failure atomicity が保証されるようにように変換する．

トランザクション単位で永続化を行うコストが大きいため，それを削減する技法が研究されている．例えば，clobber logging [20] は，トランザクション単位で，それを再実行するのに十分な変数の値だけをログに記録する技法である．障害時には，ログを元に変数の値を復元して，トランザクション再実行することでデータを復元する．

永続化の単位を，個々のトランザクションではなく，エポックに拡張する考えが Periodic Persistence [16] である．エポック長を調整することで，故障時に失われる更新量を増やす代わりに，永続化のオーバーヘッドを抑えることができる．Periodic Persistence に基づく Hashmap である Dalí[16] が，先駆的な仕事である．

Periodic Persistence の下で，ロギングのオーバーヘッドを削減する手法も数多く研究されている．Asynchronous Semantic Logging [15] は，DRAM 上のデータ構造への高水準操作（例えば insert）とその引数だけを，専用のバックグランドスレッドがログに記録する技法である．このとき，高水準操作の DRAM 上での実行が，高水準操作と引数を NVM に記録することよりも時間が掛かるとき，ロギングのレイテンシは隠蔽される．In-Cache-Line-Log (InCLL) [8] は，更新の記録対象のオブジェクトフィールドと，そのログを，NVM 上の同一キャッシュラインに配置する技法である．これにより，キャッシュラインの書き戻し（clwb）やフェンス（sflush）を発行せずに，フィールドの値とログの一貫性を保証できる．これらの仕事は，永続化の対象となるデータ構造に，ある程度仮定を置く方針を取っている．

一方，永続化対象のデータ構造に依存しない API によって，Periodic Persistence を実現する取り組みもある．本研究で拡張を加えた *CpNvm* [2][3] に加えて，Montage [18][6]，PMThreads [19]，ResPCT [13] も，それに分類される．これらのうち，ユーザスレッドを停止させずにチェックポインティングを行うのは，Montage だけである．PMThreads は，エポックの区切りとして，ユーザスレッド間の mutex lock の利用を仮定している．ResPCT は，InCLL に基づく API を与えることで，効率と汎用性を両立させた．

7　まとめ

本研究では，読み込み性能が高い永続化手法である *CpNvm* に対して double buffering と CoW を導入してプログラムを停止させない手法を提案した．本手法は data-race-free なアプリケーションでは failure atomicity を保ちながらチェックポインティング時に停止しないことを実現する．

本手法を YCSB ベンチマークで測定し，停止時間の割合はデータを永続化しない場合と比べて，2.3 倍に抑えることを確認した．さらに，*CpNvm* の強みである読み込み性能を維持することができ，スループッ

トは最低の場合でも 0.89 倍に抑えることができた．

提案手法ではまだリカバリーの実装が終わっていない．そのため，リカバリーの実装をし，リカバリーにかかる時間を既存研究と比較することが今後の課題である．

謝辞

本研究の一部は，JSPS 科研費 JP19K11904 の助成を受けたものです．また，本研究の一部は，キオクシア株式会社の支援をうけて実施したものです．

参 考 文 献

[1] : Memcached, `https://memcached.org/`.

[2] Aksun, D. and Larus, J.: Durability Through NVM Checkpointing, 12th Non-Volatile Memories Workshop (Poster), 2021.

[3] Aksun, D. T.: *Software Support for Non-Volatile Memory (NVM) Programming*, PhD Thesis, IINFCOM, EPFL, Lausanne, 2021.

[4] Azatchi, H., Levanoni, Y., Paz, H., and Petrank, E.: An On-the-Fly Mark and Sweep Garbage Collector Based on Sliding Views, *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, ACM, 2003, pp. 269–281.

[5] Cai, W., Wen, H., Beadle, H. A., Kjellqvist, C., Hedayati, M., and Scott, M. L.: Understanding and Optimizing Persistent Memory Allocation, *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ISMM '20, ACM, 2020, pp. 60–73.

[6] Cai, W., Wen, H., Maksimovski, V., Du, M., Sanna, R., Abdallah, S., and Scott, M. L.: Fast Nonblocking Persistence for Concurrent Data Structures, *35th International Symposium on Distributed Computing (DISC 2021)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 14:1–14:20.

[7] 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 Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, ACM, 2011, pp. 105–118.

[8] Cohen, N., Aksun, D. T., Avni, H., and Larus, J. R.: Fine-Grain Checkpointing with In-Cache-Line Logging, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, ACM, 2019, pp. 441–454.

[9] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R.: Benchmarking Cloud Serving Systems with YCSB, *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, ACM, 2010, pp. 143–154.

[10] Friedman, M., Ben-David, N., Wei, Y., Blelloch, G. E., and Petrank, E.: NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey, *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20, ACM, 2020, pp. 377–392.

[11] Friedman, M., Herlihy, M., Marathe, V., and Petrank, E.: A Persistent Lock-Free Queue for Non-Volatile Memory, *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, 2018, pp. 28–40.

[12] Izraelevitz, J., Mendes, H., and Scott, M.: Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model., *Distributed Computing, 30th International Symposium, DISC 2016*, Lecture Notes in Computer Science, Vol. 9888, Springer, 2016, pp. 313–327.

[13] Khorguani, A., Ropars, T., and De Palma, N.: ResPCT: Fast Checkpointing in Non-Volatile Memory for Multi-Threaded Applications, *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, ACM, 2022, pp. 525–540.

[14] Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., and Ren, J.: DudeTM: Building Durable Transactions with Decoupling for Persistent Memory, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, ACM, 2017, pp. 329–343.

[15] Memaripour, A., Izraelevitz, J., and Swanson, S.: Pronto: Easy and Fast Persistence for Volatile Data Structures, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, ACM, 2020, pp. 789–806.

[16] Nawab, F., Izraelevitz, J., Kelly, T., III, C. B. M., Chakrabarti, D. R., and Scott, M. L.: Dalí: A Periodically Persistent Hash Map, *Proceedings of the 31st International Symposium on Distributed Computing (DISC 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 91, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 37:1–37:16.

[17] Volos, H., Tack, A. J., and Swift, M. M.: Mnemosyne: Lightweight Persistent Memory, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, ACM, 2011, pp. 91–104.

[18] Wen, H., Cai, W., Du, M., Jenkins, L., Valpey, B., and Scott, M. L.: A Fast, General System for Buffered Persistent Data Structures, *Proceedings of*

the 50th International Conference on Parallel Processing, ICPP '21, ACM, 2021, pp. 73:1–73:11.

[19] 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 '20, ACM, 2020, pp. 623–637.

[20] Xu, Y., Izraelevitz, J., and Swanson, S.: Clobber-NVM: Log Less, Re-Execute More, *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, ACM, 2021, pp. 346–359.

[21] Zuriel, Y., Friedman, M., Sheffi, G., Cohen, N., and Petrank, E.: Efficient Lock-Free Durable Sets, *Proc. ACM Program. Lang.*, No. OOPSLA(2019), pp. 128:1–128:26.