DEIM Forum 2023 2b–5–3

永続メモリ向けMulti-Word Compare-and-Swap命令の改善

西村　学† 杉浦　健人† 石川　佳治†

† 名古屋大学大学院情報学研究科知能システム学専攻　〒 464–8603 愛知県名古屋市千種区不老町
E-mail: †nishimura@db.is.i.nagoya-u.ac.jp, ††{sugiura,ishikawa}@i.nagoya-u.ac.jp

あらまし　高い読み書き性能と不揮発性を両立した永続メモリが利用可能になったことで，その活用方法が注目されている．Persistent multi-word compare-and-swap（PMwCAS）命令は永続メモリ上の複数ワードをアトミックに更新できる命令であり，複雑で難しいとされる永続メモリ向けデータ構造の実装をより簡潔にすることが期待されている．しかし，現在提案されている PMwCAS 命令のアルゴリズムは確保した永続メモリ領域のガベージコレクションが必須であるなど，その実装には改善の余地がある．そこで，本研究では新しい PMwCAS 命令のアルゴリズムを提案し，提案アルゴリズムの性能について実験により評価する．実験の結果，提案手法は既存手法より常に高速であることが分かった．また，既存手法は競合が多く発生すると性能が著しく低下するのに対し，提案手法は競合の多寡によらず比較的安定した性能を示した．

キーワード　永続メモリ，ロックフリーアルゴリズム，障害回復．

1 はじめに

永続メモリと呼ばれる高い読み書き性能と不揮発性を両立したメモリが登場したことで，さまざまな分野でその活用方法の提案や研究が行われている．データベース分野においては永続メモリ向けの索引構造が多く提案されており [1, 2]，永続性の保証や障害回復といった特徴を有している．また現在では，3D XPoint 技術 [3] を使用した Intel Optane DC Persistent Memory Modules（DCPMM）が実際に永続メモリとして利用可能である．

しかしながら，永続メモリを利用するプログラムはデータの永続化を考慮する必要があり，その実装は通常よりも複雑となる．データは永続メモリより先に CPU キャッシュに書き込まれるため，書き込んだ全てのデータがすぐに永続化されるわけではない．また，永続メモリは不揮発であるため，適切にメモリを解放しないと永続的なメモリリークが発生する．そのため，プログラマはいつ障害が発生しても正しい状態に回復できるようなデータ構造の設計を求められる．

上述した問題に対して，persistent multi-word compare-and-swap（PMwCAS）命令 [4] が Wang らにより提案されている．PMwCAS 命令は永続メモリ上の複数ワードをアトミックかつロックフリーに更新できるため，永続メモリ上での実装をより簡潔にする．Wang らの PMwCAS 命令のアルゴリズムは multi-word compare-and-swap（MwCAS）命令の 1 つである CASN アルゴリズム [5] を拡張したものである．永続メモリ向けのロックフリー索引構造である Bz 木 [6] においても，データの更新や構造変更操作などに利用されている．

本研究では，より高性能な PMwCAS 命令のアルゴリズムを提案する．CASN アルゴリズムおよび Wang らの PMwCAS 命令のアルゴリズムはどちらも確保したメモリ領域のガベージコレクションを必要としており，改善の余地があると考えられる．実際に本研究グループは既にガベージコレクションを必要としない MwCAS 命令のアルゴリズム [7] を提案し，性能の改善を達成している．この MwCAS 命令のアルゴリズムを永続メモリ向けに拡張し，ガベージコレクションの不要な新しい PMwCAS 命令として提案する．そして，既存手法と比較しながら提案手法の性能について評価する．

本稿の構成は以下の通りである．まず，2 章で関連研究である永続メモリ，MwCAS 命令について概説する．次に，3 章で Wang らの PMwCAS 命令について説明し，4 章でその改善方針について述べる．そして，5 章で実験による性能評価の結果を示す．最後に，6 章で本稿のまとめと今後の課題について述べる．

2 関連研究

本章では，関連研究として永続メモリおよび MwCAS 命令について概説する．

a） 永続メモリ

揮発性メモリに匹敵する読み書き速度を持ち，より細かい粒度での読み書きが可能な不揮発性メモリを永続メモリと呼ぶ．永続メモリは従来の揮発性メモリや不揮発性メモリと異なる特性を持つため，性能を最大限に引き出すためには新しい技術が必要となる．現在，利用可能な永続メモリとして，3D XPoint 技術 [3] を使用した Intel Optane Persistent Memory（DCPMM）が存在し，いくつかの先行研究によってその性能が報告されている [1, 8–10]．代表的な特性として，従来の揮発性メモリと比べて読み書きの帯域幅が低いことが挙げられる [1]．特に，書き込みの帯域幅が強く制限される．加えて，同じキャッシュラインを連続でフラッシュするとレイテンシが非常に高くなることも分かっている [8]．したがって，DCPMM を効率的に利用するためには読み書きおよびフラッシュの回数を共に減らすことが求められる．また，永続メモリ向けのプログラミングライブ

ラリである Persistent Memory Development Kit （PMDK）[1]も公開されており，本研究のための実装においてもこれを使用する．

b） MwCAS 命令

MwCAS 命令は compare-and-swap（CAS）命令を拡張し，単一ワードから複数ワードの更新に対応させたものである．したがって，MwCAS 命令を利用すると複数の対象ワードをアトミックに更新できる．ロックフリーデータ構造は複雑な実装を要求されるため，MwCAS 命令を用いることでその実装を簡潔にすることが期待されている．Wang らの PMwCAS 命令 [4] のアルゴリズムは MwCAS 命令の 1 つである Harris らの CASN アルゴリズム [5] に基づき提案されたものである．MwCAS 命令のアルゴリズムは Harris ら以外からもいくつか提案されているが，それらはより複雑な手続きを要求するため永続化を考慮した拡張には適していない．また，本研究グループは Harris らの CASN アルゴリズムを改善しており，その結果 2 倍から 3 倍程度の性能向上を達成している [7].

3 準　　備

本章では，Wang らが提案した PMwCAS 命令のアルゴリズムについて説明する．Wang らのアルゴリズムは Harris らの CASN アルゴリズムを拡張して提案されたものであり，永続メモリ向けに各要素の永続化保証と障害回復機能を有している．

3.1 制御用データ構造

PMwCAS 命令の手続きには，実行したい CAS 命令の情報と操作の進行状況を格納した PMwCAS 記述子を使用する．PMwCAS 記述子の概観を図 1 に示す．PMwCAS 記述子は操作の始めにユーザによって割り当てられ，永続メモリ上の決められたプール領域へ常に永続化される．PMwCAS 記述子に格納された情報は手続きを進めるのに必要となるだけでなく，障害発生時の復帰処理や後述する他スレッドによる処理の補助にも利用される．

PMwCAS 記述子には次の 2 つの要素が含まれる．

a） STUTAS

PMwCAS 記述子の STATUS は手続きの進行状況を示す．以下の 4 つの値をとる．

- **Succeeded**：対象領域の予約が完了している．
- **Failed**：対象領域の予約が失敗している．
- **Undecided**：対象領域の予約が進行中である．
- **Free**：対象ワードの更新が完了している．

STATUS の値が Free の場合，その記述子を利用する PMwCAS 命令は完了しているため，そのメモリ領域は後述するガベージコレクションにより再利用される．

b） ワード記述子

PMwCAS 記述子のワード記述子は各対象ワードの情報を格納する．ワード記述子は 1 つの PMwCAS 記述子に対して複数個存在し，同時に交換するワードの個数だけ配列として格納される．ただし，図 1 においては 1 つのワード記述子しか示していない．ワード記述子には更に次の 2 つの要素が含まれる．

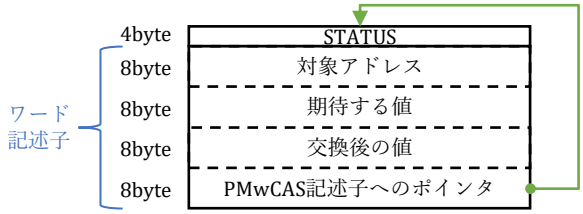

図 1　PMwCAS 記述子の構成

- 対象アドレス
- 期待する値
- 交換後の値
- 自身を格納する PMwCAS 記述子へのポインタ

3.2 PMwCAS 命令の操作手順

PMwCAS 命令の手続きは，その命令の対象となるワード領域の確保と値の更新の 2 つのステップから成る．

領域の確保は，対象ワード領域へ記述子へのポインタを埋め込むことで行う．まず，PMwCAS 記述子に含まれるワード記述子へのポインタを対象領域へ埋め込む．この手続きは実質的に restricted double-compare single-swap（RDCSS）操作であり，対象ワードが期待する値と等しいことと STATUS の値が Undecided であることを必要とする．RDCSS 操作に成功した場合，続けて同じ領域へ PMwCAS 記述子へのポインタを埋め込み，領域の確保を完了する．同様の手続きで対象となる全ての対象領域を確保できた場合，STATUS の値を Succeeded へと変更する．全ての領域の確保に失敗した場合は STATUS の値を Failed へと変更する．

値の更新は，PMwCAS 記述子の STATUS の値に応じて対象となるワード領域の値を変更することで行う．STATUS の値が Succeeded の場合，埋め込まれている PMwCAS 記述子へのポインタを対応する交換後の値へと変更する．一方で STATUS の値が Failed の場合，確保した一部の領域を元に戻すため埋め込まれている PMwCAS 記述子へのポインタを対応する元々の値（期待する値）へと変更する．値の変更後，STATUS の値を Free へと変更し PMwCAS 命令は完了となる．

3.3 制御用ビット

PMwCAS 命令の対象となる 64 bit（8 byte）のワード領域のうち，3 bit を制御ビットとして使用する．つまり，対象ワードは 61 bit で表現可能でなければならない．制御ビットには RDCSS フラグ・PMwCAS フラグ・ダーティフラグと呼ばれる 3 種類のフラグがそれぞれ 1 bit ずつ存在し，格納されているデータの種類や状態を表している．表 1 に各フラグの値（左から RDCSS フラグ・PMwCAS フラグ・ダーティフラグ）と格納されているデータとの関係を示す．なお，取り得ないフラグの組合せは省略している．

ダーティフラグはそのワードが永続化されているかどうかを表すフラグである．各スレッドはワードを埋め込む際に必ずダーティフラグを設定する．これによりダーティフラグが設定

1：https://github.com/pmem/pmdk

表 1　既存手法における各フラグの値と格納データの関係

フラグの値	格納データ
000	永続化された通常の値
001	永続化されていない通常の値
010	永続化された PMwCAS 記述子
011	永続化されていない PMwCAS 記述子
100	ワード記述子

されているワードは永続化されていない可能性があることを表し，そのワードを参照したスレッドはダーティフラグを解除しつつそれを永続化してから操作を続ける．永続化処理はオーバーヘッドが大きいため，ダーティフラグを用いることで余分な永続化を削減し効率的な永続化を実現している．

RDCSS フラグおよび PMwCAS フラグはそのワード領域に格納されているデータがワード記述子もしくは PMwCAS 記述子へのポインタであることを表すフラグである．つまり，これらのフラグが設定されている領域は既に確保された領域であることを示す．操作中，フラグの状態から他スレッドにより領域が確保されていることを発見した場合，一時的に自スレッドの処理を中断しその他スレッドの処理を補助する．このとき，埋め込まれている PMwCAS 記述子から操作のために必要な情報を参照し，手続きを進める．各スレッドでの操作は RDCSS フラグの存在により線形化され，同じ PMwCAS 命令が複数回実行されることを防いでいる．

3.4 障害回復

障害からの回復は再起動時に永続化されている記述子のプール領域を走査し，未完了の PMwCAS 命令を処理することで行う．PMwCAS 命令の進行中に障害が発生した場合，対象ワード領域に正常値だけでなく PMwCAS 記述子やワード記述子へのポインタが残ってしまう可能性がある．そのため，障害回復処理によりそれらの値を正常値へとロールフォワードもしくはロールバックしデータの一貫性を保証する．

どのように回復するかは各 PMwCAS 記述子の STATUS の値によって決定する．Undecided または Failed であれば領域の確保が完了していないか失敗しているため，対象ワードを元々の値（期待する値）へと戻す．Succeeded であれば領域の確保に成功しているため，各対象ワードを交換後の値へと変更する．それ以外であればなにもしない．その後，STATUS の値は Free に設定され，その記述子は再利用可能な状態となる．

4 PMwCAS 命令の改善

3 章で述べた Wang らによる PMwCAS 命令のアルゴリズムでは，記述子領域の動的な解放のためにエポックベースのガベージコレクションを実装している．これは，スレッド間での相互補助によりどの記述子も他のスレッドから参照される可能性があることから，操作が完了してもその時点で記述子領域を即座に解放できないためである．このようなガベージコレクションは追加の処理を必要とし，スループットとレイテンシの低下を招く．また，後追いによる処理の補助はそもそも非効率であり，これによる性能向上は期待できないと揮発性メモリ上での先行研究で示されている [7].

上述した問題はベースとなった CASN アルゴリズムから既に存在しており，本研究グループはこれを改善した MwCAS 命令のアルゴリズム [7] を提案している．この MwCAS 命令では，CASN アルゴリズムにおけるスレッド間での相互補助を排除することでガベージコレクションの不要なアルゴリズムとしている．

本章では，ガベージコレクションを用いない PMwCAS 命令のアルゴリズムを提案する．本アルゴリズムは本研究グループが提案した MwCAS 命令のアルゴリズムを拡張したものであり，永続メモリ向けにデータ構造の永続化を保証する．

表 2　提案手法における各フラグの値と格納データの種類

フラグの値	格納データ
00	永続化された通常の値
01	永続化されていない通常の値
10	PMwCAS 記述子

4.1 データ構造

提案する PMwCAS 命令においても，手続きを進めるために PMwCAS 記述子を使用する．その構成は図 1 に示す既存手法のものとほぼ同様である．ただし，STATUS が取り得る値は操作の進行中を表す InProgress，成功を表す Succeeded および完了を表す Finished のいずれか 3 つとなる．PMwCAS 記述子は配列として永続メモリ上の決められたプール領域へ静的に確保される．このとき，1 つのスレッドに対しただ 1 つの PMwCAS 記述子が割り当てられるため，使用が想定されるスレッド数に等しい数だけ確保しておけば十分である．

また，制御ビットとして PMwCAS フラグとダーティフラグの計 2 bit を使用する．RDCSS フラグは既存手法における RDCSS 操作（ワード記述子へのポインタの埋め込み）を行わないため必要としない．そのため，各ワード領域はそれぞれ 62 bit まで利用可能となり，既存手法よりも多くの値に対応する．後述する特定の条件下においてはダーティフラグも省略でき，その場合は各ワード領域が 63 bit まで利用可能となるだけでなく，必要な永続化回数の減少により性能向上が期待される．表 2 に本アルゴリズムにおける各フラグの値（左から PMwCAS フラグ・ダーティフラグ）と格納されているデータの種類の関係を示す．各スレッドはフラグの値が “00” である永続化済みの通常値を読んだときのみ自身の処理を継続し，それ以外の場合は他スレッドの PMwCAS 命令が完了するまで待機することとなる．

なお，提案手法ではスレッド間での後追いによる相互補助を行わないが，フラグ “10” において記述子自身のアドレスも対象領域へ埋め込む点に注意する．これは，記述子アドレスが後追いだけでなく障害復帰の際にも必要となるためである．フラグのみを埋め込んだ場合は障害復帰時にいずれの記述子がその領域を確保していたかが確定できないため，記述子を識別するための一意な識別子として自身のアドレスを利用する．

```
Input: desc                                   // PMwCAS 記述子
Output: boolean                               // PMwCAS が成功したら true
1  d_addr ← an address of desc with PMwCAS Flag
2  desc.status ← InProgress
3  succeeded ← true
4  persist(&desc.status)                      // STATUS の永続化
5  foreach w ∈ desc do                        // w はワード記述子
6      do
7          cur_w ← CAS(w.address, w.expected, d_addr)
8      while Descriptor(cur_w) ∨ ¬Persisted(cur_w)
9      if cur_w ≠ w.expected then
10         succeeded ← false
11         break
12 if succeeded then
13     foreach w ∈ desc do                    // w はワード記述子
14         persist(w.address)                 // 対象領域の永続化
15     desc.status ← Succeeded
16     persist(&desc.status)                  // STATUS の永続化
17 foreach embedded descriptor w ∈ desc do
18     if succeeded then v ← w.desired else v ← w.expected
19     store v with dirty flag to w.address
20     persist(w.address)                     // 耐障害性のための永続化
21     store v to w.address
22     persist(w.address)                     // 対象の交換・永続化完了
23 desc.status ← Finished
24 persist(&desc.status)                      // STATUS の永続化
25 return succeeded
```

図 2　PMwCAS 命令のアルゴリズム

4.2　実行アルゴリズム

提案する PMwCAS 命令のアルゴリズムを図 2 に示す．なお，本アルゴリズムにおいて Descriptor(x) は x が記述子であるとき，Persisted(x) は x が永続化されているときにそれぞれ真となる述語である．これらは対象領域内に存在する制御ビットの状態（PMwCAS フラグおよびダーティフラグ）から判定できる．

まず，全ての対象領域へ PMwCAS 記述子へのポインタを埋め込み，領域を確保する（5–11 行目）．このとき，対象領域に他の PMwCAS 記述子もしくは永続化されていない値が存在していた場合はスピンロックにより待機する（6–8 行目）なお，実際には待機による無駄な CPU の利用を避けるために，スピンロックでの記述子埋込みに一定の回数失敗した時点で PMwCAS 命令自体を失敗と判定しユーザに再試行を促すように実装した．全ての対象ワード領域の確保および永続化に成功した場合，記述子の STATUS を Succeeded に変更し記述子の状態も永続化する（12–16 行目）PMwCAS 命令の成功はこの STATUS 永続化が完了した時点で確定し，障害発生によりプログラムが停止したとしても後述する障害復帰処理で全ての値は交換後の値へと更新される．

その後，領域の確保に成功したかどうかに応じて対象ワードを更新し，PMwCAS 命令を完了する（17–25 行目）．成功している場合は交換後の値へ，そうでない場合は元々の値（期待する値）へ更新する（18 行目）．このとき，障害発生時におけるデータの一貫性を保証するため，まずはダーティフラグを設定した値を埋め込み永続化し（19–20 行目），続けてダーティフラグを解除した値を埋め込み永続化する（21–22 行目）．最後に，STATUS の値を Finished に変更および永続化し（23–24 行目），PMwCAS 命令が成功したかどうかの結果を返す（25 行目）．

```
Input: addr                                   // 対象ワードのメモリアドレス
Output: word
1  do
2      word ← a word from addr
3  while Descriptor(word) ∨ ¬Persisted(word)
4  return word
```

図 3　PMwCAS 命令における対象領域を読み取るアルゴリズム

PMwCAS 命令における対象領域の読み取りには，専用の読み取りアルゴリズムを使用する．そのアルゴリズムを図 3 に示す．このアルゴリズムは記述子アドレスや永続化されていない値を読み取ることを防ぐために必要となる．対象領域に記述子アドレスか永続化されていない値が存在していた場合，スピンロックにより待機する（1–3 行目）．永続化された通常の値が読み取れた場合，それを結果として返す（4 行目）．なお，実際にはスピンロックだけでなくブロッキング（スレッドの一時的な sleep）も組み合わせ，無駄な CPU の利用が発生しないよう実装した．

4.3　障害回復アルゴリズム

PMwCAS 命令の手続き中に障害が発生した場合，障害回復の手続きによりどのような状態であってもデータの一貫性を保証する．図 2 のアルゴリズムにおいて，STATUS の値と対象領域はそれぞれ表 3 に示すような状態を取り得る．データの一貫性を保証するためには，対象領域に PMwCAS 記述子が残る状態を回避し，更新前か更新後の値のみである必要がある．復帰時に永続化されている記述子領域を走査し，STATUS の値に応じて対象領域をロールバックもしくはロールフォワードすることで，適切なデータ構造を維持できる．

障害復帰時の回復アルゴリズムを図 4 に示す．この操作はプログラムの動作開始後，PMwCAS 記述子用のプール領域が確保された際に行われる．同一のプログラムであれば PMwCAS 記述子用領域は永続メモリ上の同じアドレスが利用されるため，領域上の記述子を全て走査し実行途中の PMwCAS 命令が存在するか確認する．記述子の STATUS が Succeeded であれば領域の確保まで完了している状態であるため，各対象領域に含まれる記述子アドレスを交換後の値へと変更する（3–5 行目）．InProgress であれば領域の確保が完了していないか失敗している状態であるため，各対象領域に含まれる記述子アドレスを元々の値（期待する値）へと変更する（6–8 行目）．その後，ダーティフラグの有無を確認しつつ対象領域を永続化し STATUS の値を更新する（9–15 行目）．以上の操作を全ての記述子に対して行うことで，障害発生により実行途中で終了してしまった PMwCAS 命令を完了できる．

表 3　STATUS の値と対象領域の関係

STATUS の値	対象領域の状態
InProgress	全て更新前の値
InProgress	更新前の値と PMwCAS 記述子が混在
InProgress	全て PMwCAS 記述子
Succeeded	全て PMwCAS 記述子
Succeeded	PMwCAS 記述子と更新後の値が混在
Succeeded	全て更新後の値
Finished	全て更新後の値

```
Input: desc_pool                                   // PMwCAS 記述子領域
1  foreach desc ∈ desc_pool do                     // desc_pool は記述子用プール
2      d_addr ← an address of desc with PMwCASFlag
3      if desc.status = Succeeded then
4          foreach w ∈ desc do                     // ロールフォワード
5              CAS(w.address, d_addr, w.desired)
6      else if desc.status = InProgress then
7          foreach w ∈ desc do                     // ロールバック
8              CAS(w.address, d_addr, w.expected)
9      foreach w ∈ desc do
10         v ← a current value in w.address
11         if ¬Persisted(v) then
12             store v without dirty flag to w.address
13         persist(w.address)                      // 対象領域の永続化
14     desc ← Finished
15     persist(&desc.status)                       // STATUS の永続化
```

図 4　PMwCAS 命令における障害回復アルゴリズム

4.4　ダーティフラグ排除による高速化

ダーティフラグは，永続化されていないワードが変更された場合に起こり得る障害を回避するために必要となる．ダーティフラグを使わない場合に発生する不整合の例を図 5 に示す．図 5 は，ある PMwCAS 命令が単一の対象ワードを v へ変更した直後，更に別の更新命令により v から v_{new} へ変更されたときのキャッシュと永続メモリの状態を示している．まだ永続化されていないため，永続メモリ上の対象ワードは記述子のアドレス $desc$ のままである．この時点で障害が発生した場合，PMwCAS 命令の障害回復により対象ワードは $desc$ から v へと変更されるため，v_{new} への変更が失われてしまう．ダーティフラグが存在していれば，永続化されていないワードへのアクセス（ここでは v から v_{new} への変更）を検知でき，このような不整合を回避できる．

ただし，上述した状況は PMwCAS 命令でない操作（通常の CAS 命令など）で値を変更した場合にのみ起こり得るため，値の変更を全て PMwCAS 命令で完結させる場合はダーティフラグを必要としない．ダーティフラグを不要とすれば必要な永続化回数が減るため，読み書き両方においてレイテンシの改善が期待される．図 6 は，ある PMwCAS 命令が単一の対象ワードを v へ変更した直後に，別の PMwCAS 命令による更新操作が行われたときのメモリ状態を表している．PMwCAS 命令を用い

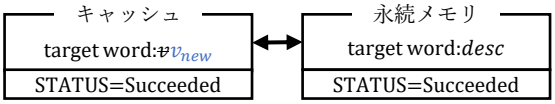

図 5　永続化されていないワードを変更したときのメモリ状態

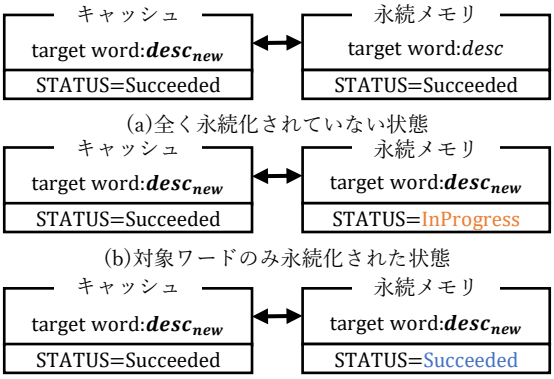

図 6　永続化されていないワードを PMwCAS により変更したときのメモリ状態

て v を v_{new} へ更新する場合，更新より前にその PMwCAS 命令の記述子 $desc_{new}$ が埋め込まれる．このとき永続メモリは記述子が永続化されるタイミングによって 3 つの状態を取り得るが，どの状態で障害が発生しても正しく復帰可能である．図 6(a) は記述子が全く永続化されていない場合であり，$desc$ を v へ変更することで復帰する．図 6(b) は対象ワードのみ永続化された場合であり，STATUS が InProgress であることから $desc_{new}$ を v へ戻すことで復帰する．図 6(c) は記述子が全て永続化された場合であり，$desc_{new}$ を v_{new} へ変更することで復帰する．いずれの場合も v_{new} への更新は完了しないもしくは成功する結果となり，不整合は発生しない．つまり，記述子の埋込みおよび永続化がダーティフラグに代わって線形化ポイントとなり，障害発生時の一貫性を保証する．

5　性能評価

本章では，実験による提案手法の性能評価について述べる．実験に使用したサーバの構成を表 4 に示す．なお，本構成は NUMA に対応しているため 2 つの CPU ノードが存在するが，永続メモリを管理するまとまり（メモリプール）は各ノードに分けて生成したため，CPU ソケット間のリモートアクセスを避けるために片方の CPU のみを用いるよう設定して測定した．また，Wang らの PMwCAS 命令として microsoft/pmwcas のオープンソースライブラリ[2]を利用した．

5.1　実験用ベンチマーク

MwCAS 命令向けのベンチマーク[3]を拡張し，PMwCAS 命令の性能を評価するためのベンチマークを C++で実装した．ベン

2：https://github.com/microsoft/pmwcas
3：https://github.com/dbgroup-nagoya-u/mwcas-benchmark

表 4　実験用サーバの構成

CPU	Intel(R) Xeon(R) Gold 6258R (two sockets)
RAM	DIMM DDR4 (Registered) 2933 MHz (16GB × 12)
OS	Ubuntu 20.04.5 LTS
Compiler	GNU C++ ver. 9.4.0

チマークは以下のように動作する．まず，8 byte のワード 100 万個からなる配列を永続メモリ上に用意し，全て 0 で初期化する．次に，配列上のワードを与えられた数だけジップの法則 [11] にしたがってランダムに選択する．そして，選択されたワードに対して，現在値に 1 だけ加算するような PMwCAS 命令を成功するまで実行する．PMwCAS 命令は 1 スレッドあたり 100 万回実行される．このワークロードの結果として，スループットとパーセンタイルでのレイテンシ（5 パーセンタイル刻み）を出力する．パーセンタイルは，計測した値を昇順で並べたときにある値がどこに位置するかを示す．例えば，レイテンシを 100 個だけ計測したとき，99 パーセンタイルレイテンシは小さいレイテンシから数えて 99 番目に位置するレイテンシである．

ベンチマークにはパラメータとして同時実行スレッド数，一度に交換するワードの個数，交換する対象ワードの偏り具合を指定する．偏り具合（skew パラメータ）は以下のジップの法則の式における α の値で制御し，大きいほど偏りが強くなり多くの競合が発生する．

$$f(k;\alpha,|W|) = \frac{1/k^{\alpha}}{\sum_{n=1}^{|W|} 1/n^{\alpha}} \quad (1)$$

測定においては skew パラメータが 1 であるとき（$\alpha = 1$）のワークロードを競合の多い環境，0 であるとき（$\alpha = 0$）のワークロードを競合の少ない環境とした．

ベンチマークを用いて測定を 3 回繰り返し，それらの平均を測定結果とした．測定結果における凡例は，それぞれ以下の実装を用いたことを表す．

- **microsoft/pmwcas**：既存手法（Wang らの手法）
- **PMwCAS-DF**：提案手法（ダーティフラグを用いる）
- **PMwCAS**：提案手法（ダーティフラグを用いない）

なお，既存手法による実験は一部完了しない場合があったため，それらは結果から省略している．

5.2 実験結果

各種パラメータを変化させた際の結果と得られた知見について述べる．なお，k 個のワードを交換する際の PMwCAS 命令について，適宜 PkwCAS と略記する．例えば，3 ワードの PMwCAS 命令は P3wCAS と表記する．

5.2.1 提案手法と既存手法との性能比較

図 7 および図 8 は P3wCAS を競合の少ない環境で実行したときのスループットとレイテンシを示している．これらの結果から，提案手法は既存手法に比べ高速であることが分かる．競合の少ない環境では既存手法におけるスレッド間での相互補助がほとんど行われないため，相互補助のための追加要素やガベージコレクションが性能の妨げになっていると考えられる．

また，提案手法においてもダーティフラグを排除することで

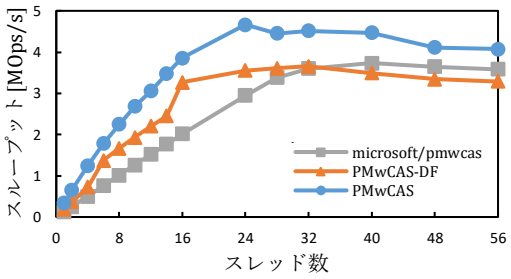

図 7　低競合環境における P3wCAS のスループット

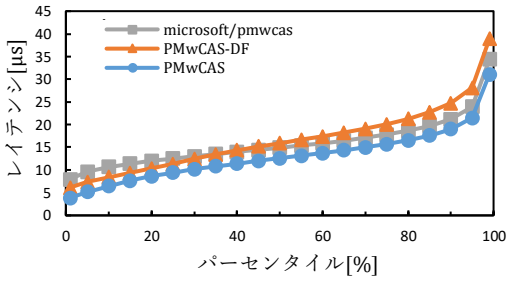

図 8　低競合環境における P3wCAS のレイテンシ（56 スレッド利用）

性能が向上することが確認できた．特に，ダーティフラグを用いる提案手法はスレッド数が多い時に既存手法よりもスループットが低くなり，テールレイテンシも高くなっている．スレッド数が多い環境ではフラッシュ回数の増加により永続メモリの書き込み性能の上限に達しやすくなり，性能が低下したと考えられる．

ただし，いずれの手法も物理コア数である 28 スレッドを超えて性能がスケールしていない点に注意する．永続メモリは揮発性メモリに比べ読み書きのレイテンシが大きいため，永続化のためにフラッシュ命令が必要となる PMwCAS 命令ではハードウェア性能に律速されやすいためである．

図 9 および図 10 は P3wCAS を競合の多い環境で実行したときのスループットとレイテンシを示している．これらの結果においても，提案手法は既存手法を上回ることが分かる．特に，既存手法は競合の発生時に相互補助による早期の命令完了を意図した手続きとなっているにもかかわらず，性能が著しく低下した．これは，永続メモリの読み書き性能が揮発性メモリのそれよりも低い点が影響したと考えられる．他スレッドの PMwCAS 命令を補助するには対応する PMwCAS 記述子を読み込む必要がある上，補助された元々のスレッドは補助されたという結果を認識するための読み出しも必要となる．つまり，速度の遅い永続メモリを介した情報のやり取りが多発したことで既存手法の性能が大きく低下したと考えられる．

一方で，提案手法は PMwCAS 命令を開始したスレッドがその手続きを必ず完遂するため，永続メモリへの読み書きは比較的少ない．その結果，80 パーセンタイル以降のレイテンシに表れているように競合による待ち時間自体は発生しているが，多くの PMwCAS 命令はその操作を競合の少ない環境と同程度のレイテンシで実行できている．特に，ダーティフラグを用いない提案手法は 80 パーセンタイルを境に大きくレイテンシの値

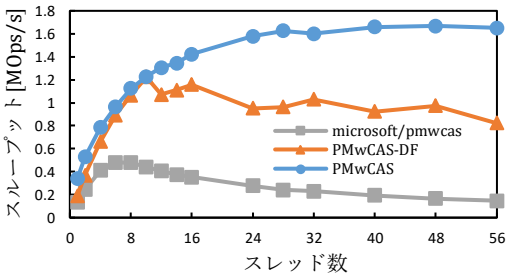

図 9　高競合環境における P3wCAS のスループット

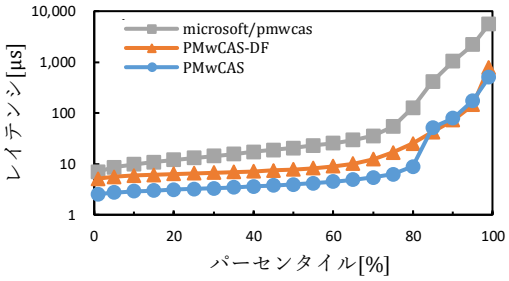

図 10　高競合環境における P3wCAS のレイテンシ（56 スレッド使用）

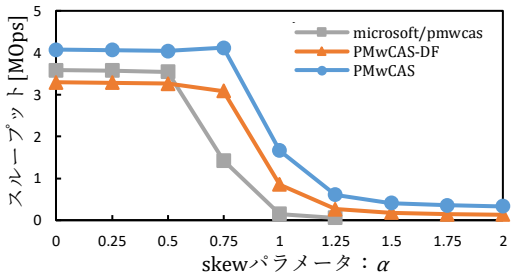

図 11　P3wCAS において偏りを変化させた際のスループット（56 スレッド使用）

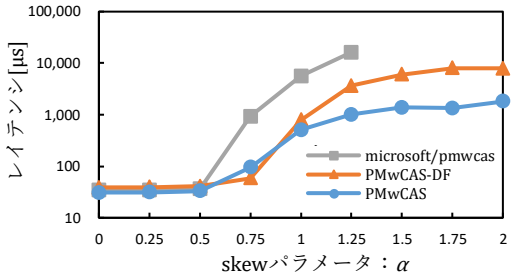

図 12　P3wCAS において偏りを変化させた際の 99 パーセンタイルレイテンシ（56 スレッド使用）

が変化しており，ダーティフラグの排除により 1 回あたりの処理時間を削減したことで競合の発生自体が抑えられていることがわかる．

5.2.2　対象ワードの偏りが性能に与える影響の評価

図 11 と図 12 は交換対象の偏りの程度を変えながら 56 スレッドで P3wCAS を実行したときのスループットと 99 パーセンタイルレイテンシを示している．提案手法は既存手法をほとんど上回る性能を示したものの，いずれの手法においても競合の増加に伴い性能が大きく低下することがわかる．これは，競合が多く発生すると対象が操作中である可能性が高く，その場合に待機しなければならない時間が増加したためだと考えられる．特に，ダーティフラグを用いる提案手法は必要な永続化回数が多く，より多くの待ち時間が発生し競合の発生頻度も増加する．

偏りが小さいとき，ダーティフラグを用いる提案手法は既存手法よりスループットが低くなった．これは，追加で必要となる永続化処理にかかる時間が競合の程度によらず大きいためだと考えられる．一方で，偏りが大きくなるにつれ，既存手法の特にレイテンシが大きく増加していることがわかる．これは前述したようにスレッド間での相互補助によるものだと考えられ，提案手法では偏りに対するレイテンシの増加がある程度抑えられているのに対し，既存手法では特に skew パラメータを 1 より大きくした際の実行が極めて遅くなった．提案手法では対象ワード領域に他の PMwCAS 記述子が埋め込まれているとき，一定回数のスピンロック試行後に埋込みが成功しなかった場合はその PMwCAS 命令を失敗扱いとしてユーザに結果を返却する．その後，再試行のために同領域の値を改めて読み出す際はスピンロックとブロッキング（スレッドの一時的な sleep）を組み合わせることで，あるスレッドが無駄に CPU を利用しないよう実装してある．そのため，偏りを極めて強くしてもいずれかのスレッドが順次 PMwCAS 命令を完了させ，性能を維持できたと考えられる．

5.2.3　対象ワード数が性能に与える影響の評価

図 13 と図 14 は競合の少ない環境で交換するワード数を変えながら PMwCAS 命令を実行したときのスループットと 99 パーセンタイルレイテンシを示している．これらの結果から，いずれの手法も交換ワード数の増加に伴い性能が低下することが分かる．特に，交換ワード数が少ないうちはダーティフラグを用いない提案手法が既存手法よりも良い性能を持つのに対し，ワード数を増やすにつれその性能差が縮まっていくことが分かる．これは，交換ワード数が少ないうちは手続きとしての簡潔さが性能に影響を及ぼしているのに対し，ワード数が増えるにつれ永続メモリの読み書きがボトルネックの大半を占めるようになったためだと考えられる．また，ダーティフラグを用いる提案手法は交換ワード数が多い際に既存手法よりも劣るが，追加のフラッシュ命令が必要なためだと考えられる．

図 15 と図 16 は競合の多い環境で交換するワード数を変えながら PMwCAS 命令を実行したときのスループットと 99 パーセンタイルレイテンシを示している．この結果から，少ないワード数においても既存手法におけるスレッド間での相互補助が有効でないことが分かる．対象ワードが 1 つだけ，つまり P1WCAS の場合，スレッド間での相互補助は読み取った記述子に応じて目的の値へと交換する作業のみとなる．複数ワードの交換を行う場合に比べ永続メモリへのアクセス回数は最も抑えられるが，この環境でも永続メモリを介したデータのやり取りが大きなボトルネックとなることが確認できた．

一方で，ダーティフラグの排除がワード数の増加に対する性能の頑健さに寄与していることも確認できた．ダーティフラグを用いない提案手法では用いる場合に比べて 99 パーセンタイ

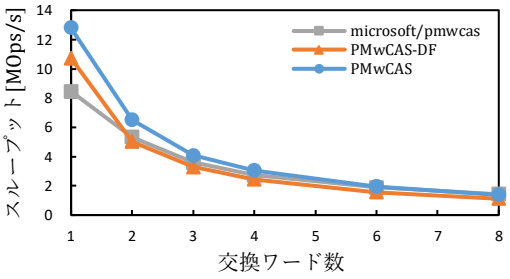

図 13　低競合環境で交換ワード数を変えた際のスループット（56 スレッド利用）

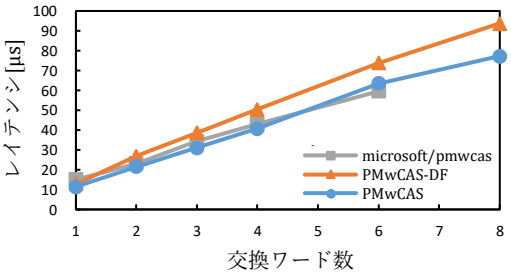

図 14　低競合環境で交換ワード数を変えた際の 99 パーセンタイルレイテンシ（56 スレッド利用）

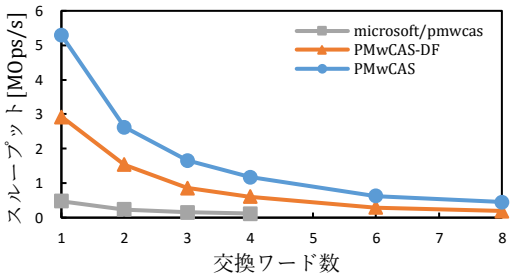

図 15　高競合環境で交換ワード数を変えた際のスループット（56 スレッド利用）

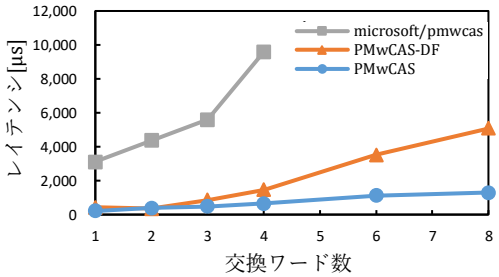

図 16　高競合環境で交換ワード数を変えた際の 99 パーセンタイルレイテンシ（56 スレッド利用）

ルレイテンシの増加がかなり抑えられており，多数のワードの交換においても安定したスループットとレイテンシを達成している．これは交換するワード数が増加したことで競合の発生確率も増加したためであり，必要なフラッシュ命令を削減したことによる 1 命令あたりの処理時間削減が有効に働いたと考えられる．

6　おわりに

本研究では，新しい PMwCAS 命令のアルゴリズムを提案し，その性能をいくつかの実験により評価した．このアルゴリズムは，本研究グループが提案したガベージコレクションの不要な MwCAS 命令のアルゴリズムを永続メモリ向けに拡張したものである．実験の結果，提案手法は既存手法より常に高速であり，競合の多寡に関わらず安定した性能を発揮できることを示した．また，永続メモリへの書き込みの増加が性能の低下をもたらすことを示した．

今後の課題は，提案した PMwCAS 命令のさらなる性能評価である．永続メモリにおける PMwCAS 命令の有効性を示すためには，ロックフリーキューなどの他のロックフリーデータ構造でも検証が必要である．また，本研究の結果は NUMA を考慮したものではないため，リモートアクセスが発生する場合の性能についても評価する予定である．

謝　辞

本研究は JSPS 科研費 JP20K19804，JP21H03555，JP22H03594 の助成，および国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務（JPNP16007）の結果得られたものである．

文　献

[1] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, “Evaluating persistent memory range indexes,” *Proc. VLDB*, vol. 13, no. 4, pp. 574–587, 2019.

[2] Y. He, D. Lu, K. Huang, and T. Wang, “Evaluating persistent memory range indexes: part two,” *arXiv preprint arXiv:2201.13047*, 2022.

[3] R. Crooke and M. Durcan, “A revolutionary breakthrough in memory technology.” Intel 3D XPoint launch keynote, 2015.

[4] T. Wang, J. Levandoski, and P.-A. Larson, “Easy lock-free indexing in non-volatile memory,” in *Proc. ICDE*, pp. 461–472, IEEE, 2018.

[5] T. L. Harris, K. Fraser, and I. A. Pratt, “A practical multi-word compare-and-swap operation,” in *International Symposium on Distributed Computing*, pp. 265–279, 2002.

[6] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, “BzTree: A high-performance latch-free range index for non-volatile memory,” *Proc. VLDB*, vol. 11, no. 5, pp. 553–565, 2018.

[7] K. Sugiura and Y. Ishikawa, “Implementation of a multi-word compare-and-swap operation without garbage collection,” *IEICE Transactions on Information and Systems*, vol. 105, no. 5, pp. 946–954, 2022.

[8] A. Van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, “Persistent memory i/o primitives,” in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pp. 1–7, 2019.

[9] S. Gugnani, A. Kashyap, and X. Lu, “Understanding the idiosyncrasies of real persistent memory,” *Proc. VLDB*, vol. 14, no. 4, pp. 626–639, 2020.

[10] L. Benson, L. Papke, and T. Rabl, “Perma-bench: benchmarking persistent memory access,” *Proc. VLDB*, vol. 15, no. 11, pp. 2463–2476, 2022.

[11] 長尾真, “岩波講座ソフトウェア科学 15 自然言語処理,” **岩波書店**, pp. 180–181, 1996.