Vol.2021-OS-151 No.13
2021/3/2

DBMS 集約環境における Persistent Memory の動的割り当て機構

十文字 優斗[,a)]　山田 浩史[1,b)]

概要：PM(Persistent Memory) を利用する手段として PMDK やファイルシステムの対応が進み, アプリケーションはこれらを通して活用が広がっている. 一方で, 確保手段が限られている関係上制約があり, 特に確保容量の柔軟性が欠けている. PMDK では事前に確保した領域に対して処理を行う関係上容量を容易に増減することができない. これは複数のアプリケーションが単一マシン上で PM を共有する際には大きな欠点となる. 事前に割り振ったメモリ領域ではその後のワークロード変化に対応することができないため, 負荷が増えた際にもその設定で動作させることになる. 本研究では複数 DBMS 間でワークロードの変化に応じて PM の割り当て量を調整することで性能の最大化を目指す.

1. はじめに

Persistent Memory(PM) は不揮発性を備えたバイトアドレッサブルなメモリである. PM は HDD や SSD と比較して高いスループットや帯域に加え低いレイテンシで書き込みと読み出しが可能であり, DRAM と比較すると再起動後までデータを保持する永続性を持ち容量もより大きいデバイスである. 加えて PM は DRAM と同じくアドレス単位での読み書きが可能であり, その特性上ブロックデバイスへ特化したデータ構造でなくても保存が容易であるなどの利点がある. 実際に PM は製品化されており, Intel Optane DC Persistent Memory [1], [2] と呼ばれる PM が市場に出ている. また, Google Cloud Platform [19] に採用されるなど, クラウド環境においても実際に PM が利用され始めている.

PM の登場により, データベース管理システム (DBMS) も大きくその恩恵を受けるアプリケーションであり, PM の活用を前提として DBMS 向けのソフトウェアメカニズムが多く提案されている [16], [17], [23], [27], [28], [43], [48]. PM の大容量性は DBMS のメモリ容量を増加させ, 高速なストレージとしての側面はストレージ I/O 時間の短縮によるレイテンシの削減に大きく寄与する [8]. たとえば MyNVM [15] のように DRAM 容量を拡張することで DBMS としての性能を向上させることが確認されており [40], [51], Espresso [47] のようなフレームワークでの PM 対応や SLM-DB [27] のようにデータ構造での対応によって DBMS の性能向上手法が提案されている. また, 実際に pmem-rocksdb [37] などの PM を利用する DBMS がオープンソース化されている.

しかしながら, これらの DBMS は PM を独占的に利用することを前提としており, DBMS を集約した際に柔軟に PM を割り当てることが難しい. 現行の PM 向けの DBMS は利用する PM の容量を起動時に決定し, 稼働中のサイズ変化を想定していない. そのため, DBMS を集約した際にワークロードの変化に追従しながら PM の割り当てを動的に変更することができない. 結果として, クラウド環境において PM を効率的に活用することができず, DBMS の集約率低下やワークロードへの追従が困難となってしまう.

本研究では, PM 向け DBMS が PM の割り当て量によってそのスループットが変動することを示し, PM の動的な割り当てを可能にする機構を提案する. 本研究では, PM 向けに改良された RocksDB [7] である pmem-rocksdb をケーススタディとし, Memtable, SSTable, Write-ahead Logging それぞれに PM を適用した場合に, PM のサイズを変更することによってスループットが変動することを定量的に示す. また, pmem-rocksdb の負荷を監視しながら, これらのサイズを動的に変更する機構についても述べる.

本論文の構成は次の通りである. 第 2 章で DBMS に対する PM の利用に関する実態と発生する DBMS 集約下での効率低下の問題点について説明する, 第 3 章では PM を利用する環境についての既存研究について問題点の状況を確

[1] 情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，東京農工大学
Presently with Tokyo University of Agriculture and Technology
[a)] jumonji@asg.cs.tuat.ac.jp
[b)] hiroshiy@cc.tuat.ac.jp

ⓒ 2021 Information Processing Society of Japan

認し, 第 4 章において問題解決のための PM 利用手順の提案を述べる. 第 5 章では提案手法に基づき pmem-rocksdb をもとに PM 活用手順と問題解決手順の設計手法を説明し, 第 6 章では現在実際に PM 活用時に性能の改善が見込めることを実験を通して最後に, 第 7 章にて本研究の課題についてまとめる.

2. 背景

2.1 Persistent Memory

PM は再起動後までその内容が保持されるという機能を持ち, DRAM と同等な主記憶装置としての利用が可能なハードウェアである. PM は DRAM と異なり 3D-XPoint [25] や Phase-Change Memmory [41] によって電源が途切れたのちも内容が維持される. 現在, PM は Intel Optane DC Persistent Memory として実際に搭載可能なハードウェアが入手可能になっており, その容量の大きさや永続性を活用する手段が研究されている. 容量の点では大容量な DRAM として DBMS のキャッシュとして読み出し速度の向上に貢献している. ストレージとしても HDD や SSD ではアクセス時間が 10 ms 程度かかるのに対して PM では 100 ns 程度であることから一部またはすべてのストレージ領域を PM に配置することで高速な書き込みも実現可能となっている [46].

PM はこのほかに DRAM と同様なアドレス単位でのアクセスが容易であり, ランダムアクセス性能が高く, ワード単位での細かいアクセスが可能となっている [22]. このようなアクセスは HDD や SSD のようなブロックデバイスとは異なる書き込み手順が必要となり, メモリへの書き込みの永続化手段が PMDK のようなライブラリやフレームワークを用いて実装されている. 書き込みや読み出し自体には通常の DRAM で利用する命令が利用可能であり, ライブラリ等を利用することでキャッシュなどをあまり意識せずに PM を利用するコードが記述可能である. ただし, これらの利用方法ではアプリケーションへの大きな変更が必要となることが多く複雑であることから [36], PM を利用した FS を用いて既存のファイル操作を高速にする利用方法も取られている. FS としては ext4 や xfs が PM に対応しており, それぞれ PM 上にパーティションを作成し通常のファイルと同様に操作が可能であり, ファイルをメモリ上にマップした際には DAX によるダイレクトな書き込みが行われる [21], [50].

2.2 DBMS に対する PM 利用

現在の DBMS は CPU や SSD などのストレージの高速化に伴ってレイテンシが重要となっており, PM は重要なハードウェアとなっている. 既に PM を利用する DBMS として pmem-rocksdb [37], memcached-pm [14] や Redis-pmem [13] のようなアプリケーションが実装されており, PM のバイト単位でアクセスできる利点を活用して更なる性能の向上を達成している [10]. ただし, これらの実装では PM に対応したデータ構造の設計と複雑な実装が必要となり実装は困難となる [42], [44] ため, 代替手段として PM に対応した FS を利用する手段がある. 複雑な実装を行わなくても SSD 等のストレージに保存していた内容を PM に対応した FS を通して保存するだけでも書き込みおよび読み出し性能の向上につながる [11], [33].

既存の DBMS に対してに PM を利用する際に PM 上に配置するデータ構造やファイルは複数の選択肢がある. 以下に, LSM-tree [35] データ構造を利用した key-value store である RocksDB [7] に PM を適用する例を示す.

2.2.1 MemTable に対する PM 利用

LSM-tree は LevelDB [20] や Apache HBase [9] などで広く利用されるデータ構造で, 書き込み要求されたデータを一度メモリ上に保持し, 一定容量溜まってからディスク上に書き出す構造となっている [32]. ここで一度メモリ上に保持されるデータ構造が MemTable であり, この段階ではランダムアクセスが容易な skiplist [39] によって key から value が得られる構造となっている. MemTable に書き込まれたデータはこのままでは永続性がないため WAL(Write ahead Logging) を併用することでデータの永続化を行っている. このため, MemTable を PM 上に配置し永続性を得ることで WAL 書き込み処理を省き, 書き込みレイテンシの削減が可能である. MemTable はメモリ上に配置することを想定していることから FS を利用した PM 利用手段は利用できず, PMDK [6] を利用することでコードの変更を抑えながら PM 上にデータを配置することを可能にしている. 実際に pmem-rocksdb では PMDK を利用して MemTable の一部を PM 上に配置しており, これによって WAL 処理を短縮する.

2.2.2 WAL に対する PM 利用

MemTable 自体を PM 上に配置するにはコードの変更量が多くなるため, FS 上に構築された WAL を PM に配置する手段もある. RocksDB では WAL いくつかのファイルとして構築されるため, PM に対応した FS を用いたパーティションを用意することでコードの変更をほとんどまたは全く行うことなく PM に対応できる. PM 上に配置された WAL はブロックデバイスに最適化された書き込み手順を踏むことになるため最大の効率を得ることはできないが, SSD 等のデバイスよりも低いレイテンシで書き込みが完了するため DB 全体の書き込みレイテンシの削減に貢献する.

2.2.3 SSTable に対する PM 利用

LSM-tree において MemTable の容量が一定に達した時点で内容をソートした状態で SSTable という一つのファイルを書き出す. SSTable はいくつかのレベルに分けられ MemTable はまず Level 1 の SSTable としてファイルに

ⓒ 2021 Information Processing Society of Japan

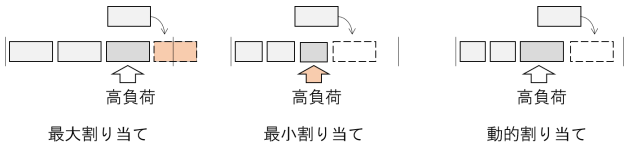

図 1 複数 DBMS に対する PM 割り当ての問題

書き出される．ある Level のファイル容量が設定された閾値を超えると各ファイル内容をマージして次の Level の SSTable として書き出される．SSTable への書き込み性能は全体のスループットに影響を与えるが PM に配置することで性能を向上する．SSTable はブロックデバイスを前提とした一定サイズでの書き込みを前提としていることから, PM 上に配置するには PM に対応した FS を利用することが現実的である．

2.3 複数の PM 利用 DBMS における問題

PM 利用環境では DBMS 集約下で PM の割り当てについて柔軟性が低下し, 性能を最大限に発揮できない状況となる．各 DBMS が異なる用途で PM を利用する環境も想定されるが [49], それぞれが固定容量の割り当てを要求するため新たに割り当ての制約となる．各 DBMS が最大量の割り当てを実施していれば割り当て可能 DBMS 数が低下し, 最小限の割り当てでは高負荷に耐えられない DBMS が出てくる (図. 1)．前述したように PM ではライブラリ等を用いたデータ構造の構築や FS を利用したファイルの配置によってレイテンシの低減やスループットの向上が達成される．しかし, これらの手法には許容される容量上限を容易に切り替えられないうえ, 上限を課されることによってワークロードの変化に対応できず性能の向上も十分に達成されない．

PMDK での利用方法ではコンパイル時に指定した容量を確保する手法 [38] や FS を通してファイル同様に truncate できるマップされた領域を得る手法がある．前者の手法ではコンパイル時点で使用可能な容量の上限が決まってしまう上実行中に割り当てを減らすことも不能となる．後者の場合においてはデータ構造によってその結果は異なってくるが, 現状のデータ構造では割り当て容量の変更を想定しておらず削減も拡張も不能である．また, FS を利用した手法ではファイルサイズを削減できる構造である必要があり, 通常ファイルを想定した構造では PM 上のファイル使用量のみを変化させることが考慮されていない．

容量の変更ができないことにより, 本来より多く割り当てることでスループットやレイテンシの改善が可能な DBMS に対して割り当てを増やすことができず, 負荷が小さく PM の割り当てがほとんど不要な DBMS に対しても常に一定量の割り当てが占有された状態となる．PM をメモリとして利用する手法でも DRAM の拡張として利用する手法とストレージとして利用する手法では ndctl [5] や ipmctl [3] を用いて namespace やパーティションによって事前に領域を分割する必要があり, DBMS の実行中にこれらの割り当て量や範囲を変更することは困難である．また, 一度割り当てられた領域が解放されない手段が多数あるため, 実行される DBMS 数に応じて割り当てられる PM 領域量が増加し, 各 DBMS に許容される割り当て量は限られたものとなる．PM はその容量からも多数の用途で同時に利用されることが想定されており, DBMS が集約された環境が現実的である以上割り当て量の増減が容易でないことは全体の性能の最適化の課題となる．

3. 関連研究

PM を利用する DBMS や手段に関する技術は数多く研究されている．本章では PM を利用する手段とそれを利用した DBMS またその構造についての既存研究について, PM 割り当ての柔軟性が低く影響を及ぼすかことについて説明する．

3.1 PM フレームワーク

PM を利用する手段としてはライブラリやフレームワークを利用する手法があるが, これらは割り当て領域の事前確保や PM の確保領域の管理をアプリケーションで行えないことにより割り当ての調整が困難となっている．PM 利用手段となるライブラリとして Intel から提供されている PMDK があるが, PMDK で提供される API では主にプール領域の容量をコンパイル時に指定する利用方法と FS を通して拡大縮小可能な領域を確保して PMDK に管理を任せる方法がある．FS を利用していても利用する容量の管理は不能であるため, PMDK を利用した手法では確保領域の拡張は容易であっても削減によって他プロセスの利用可能量を増やすことはできない．Kevlar [18] は PM を管理するフレームワークでありページ単位での PM 領域の割り当てを管理可能であるが, 未使用領域の回収については考慮していない．

Espresso [47] や AutoPersist [43], go-pmem [16] では Java と Go の言語ランタイムに PM を管理する機構を加え実装難易度を下げているが, アプリケーションからの PM 管理が不能であり PM 割り当ての管理が困難である．これらの手法では言語ランタイムが持つガベージコレクタを中心としたメモリ管理機構があるため PM 上のオブジェクトとそれ以外のオブジェクトを異なる領域に配置しているが, 実際に確保される PM 上のメモリ空間はメモリ管理機構にゆだねられている．このため特に JVM ではプロセスが稼働している間一定量のメモリ空間が占有される．go-pmem では PM に確保される領域の拡張を考慮しており, 削減については実装されておらず PM 割り当ての競合は免れない．Java Persistence API [34] や Persistent Collections for Java [24] においては事前の確保がないものもあるが能

動的な解放が考慮されていない.

3.2 既存 DBMS の PM 対応

NVMcached [48] や NVMRocks [31] は既存の DBMS をもとに PM を適用した例であるが, これらの手法は既存の DRAM に対するデータ構造をもとにしており PM 領域に対応するデータ構造のサイズを変更することを考慮していない. これは DRAM 上に配置したデータ構造であればそれ以外に退避させる対象が存在しないことが原因である. 既存 DBMS を利用することで既存の最適化や性能向上手段が活用できるが [29], データ構造の設計段階で PM という階層が考慮されていない. PM では配置しきれないデータを DRAM に配置することや他のストレージに配置することを考慮する必要があり, この点でこれらの手法は制約が課せられている.

3.3 PM 対応データ構造

PM は既存のデバイスと異なる設計目標が必要な関係上新規のデータ構造も考案されている. μ Tree [12] や FAST&FAIR [23], Recipe [30] は PM を SSD などと同様にストレージとして利用することを前提とした設計のデータ構造やインデックスである. これらの手法は PM のアドレス単位のアクセスが可能である点や SSD より高速な読み書きが可能である点を最大限に活用できる手段であるが, この方法では一度書き込まれたデータは削除されるまで PM 上の領域を占有し続けるためほかのアプリケーションが利用可能な PM 領域を減らし続ける一方である. SLM-DB [26] では PM と SSD 両方を利用することで PM をバッファやキャッシュとして利用する手法であり, PM によりスループットとレイテンシの改善を図っている. このように PM と SSD を併用する際には PM 内のデータの退避場所に SSD を選ぶことが可能となるが, データ構造としては一定領域の PM を自由に利用できる前提となっているため容量の解放は不能である. Nove-LSM [27] や 3Tier-BM [45] においても PM をディスクとメインメモリの中間に配置しバッファとして利用するが, この場合においても容量を最大限利用することを考えており実行中での容量の削減は考慮されていない.

4. 提案

本研究では単一マシン上の複数の DBMS 間で協調し PM の割り当てを調整する機構を提案する. 複数の DBMS における最適な状態を全体のスループットの合計が最大となる状態とする. 対象とする PM の利用手段については RocksDB に対する MemTable, WAL, SSTable を PM に配置する手法とする. 各 DBMS は自プロセスの負荷を通知し, 負荷に基づいて設定された PM 割り当てに従って PM 使用量の上限を設定する. 本手法は DBMS における使用量制限管理と PMDK による PM 割り当て管理によって全体のスループットを最大化する.

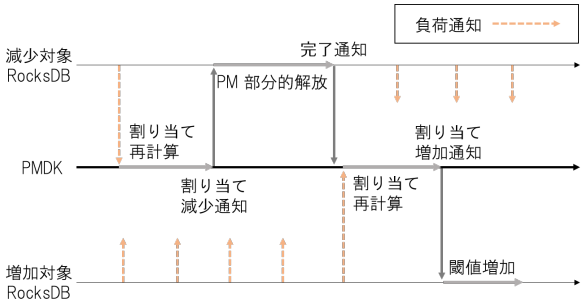

図 2 PMDK 負荷通知手順

4.1 PM 使用量制限

MemTable, WAL, SSTable の各対象について DBMS の設定を利用して PM の使用量に制限を与えた状態で PM 上にデータを配置する. 以下に各対象に対する PM への配置手法とその手法での容量制限について説明する.

4.1.1 MemTable

MemTable を対象とした手法では pmem-rocksdb を利用して PMDK を通して MemTable 内の key および value を PM に保持する. MemTable は skiplist 構造を持つがこの構造自体は再起動時に直前の状態を復元するために重要な内容ではないので key-value のみを PMDK の libpmemobj [4] を利用して保存する. libpmemobj ではオブジェクトという単位でメモリ領域を確保し内容の読み書きが可能となる. この手法では PM の使用量は MemTable の容量上限によって設定可能であり, 容量上限が十分に大きい値であればまとまったデータをディスクに書き出すことができるためスループットの向上につながる.

4.1.2 WAL

WAL は RocksDB ではファイルとして保存されるため PM に対応した FS を通して PM 上に配置する. 配置された WAL ファイルは新たなログを追加するたびに追加書き込みが行われるが SSD よりも高速に完了するため書き込みレイテンシが向上する. WAL の PM 上の容量を制限するには WAL ファイルサイズ上限を設定する必要があり, この値が小さくなると MemTable に加えられる変更回数が減るため MemTable の容量制限と同様にスループットの制限につながる.

4.1.3 SSTable

RocksDB の SSTable のうち Level 1 に該当する SSTable ファイルのみを PM 上に配置する. SSTable 全てを PM 上に配置しては容量の上限が設定不能であるのに対し, MemTable を書き出す SSTable 飲みであれば Level 1 ファイル全体の上限値を制限することで PM の使用量を制限できる. また, MemTable を書き出す SSTable の書き出し時間が短縮されれば全体のスループットの向上につながる.

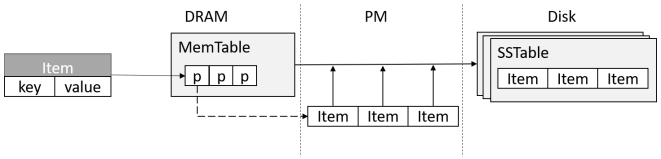

図 3 MemTable の PM 配置設計

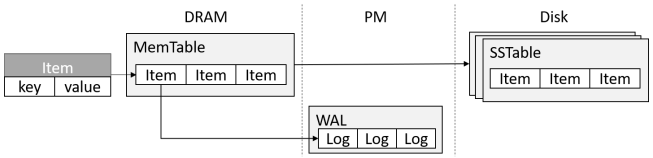

図 4 WAL の PM 配置設計

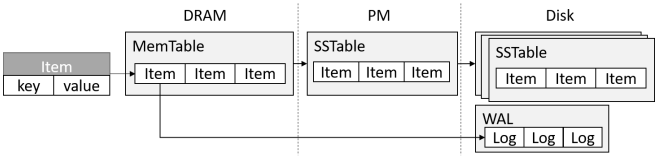

図 5 SSTable の PM 配置設計

4.2 PM 割り当て管理

本手法では各 DBMS 間の PM の使用量上限を PMDK で管理する. PMDK には常に各 DBMS の負荷を通知させることで最適な割り当てを計算し, これをもとに各 DBMS に割り当て量の増減を通知する. 割り当て量の増減が発生した DBMS はそれに従って PM 使用量を制限する設定を変更し PMDK に完了を通知する. PMDK では上限が減少する対象に先に通知を行い設定値の減少を確認次第, 増加対象に割り当ての増加を通知する (図. 2).

5. 設計

本研究では RocksDB および PMDK に対して本章では対象とする MemTable, WAL, SSTable に対する設計方針を述べる.

5.1 MemTable

MemTable に pmem-rocksdb の実装を利用して PM 上に MemTable を配置する (図. 3). pmem-rocksdb では MemTable において key および value を PM 上に保存し, skiplist 上にはそのオブジェクトへのポインタのみを保存する. PM 上へのオブジェクトの配置には libpmemobj を利用して配置する.

このため, MemTable が占有する PM 容量は MemTable のメモリ使用量となり, MemTable の容量変更によって PM 使用量を変更する. PM の割り当て容量を削減する場合は MemTable の内容を SSTable に書き出すことで PM を開放する. PMDK では容量減少を RocksDB に要求したのち, MemTable が解放されたことを確認次第増加の必要な別プロセスに容量増加を通知する.

5.2 WAL

WAL は PM に対応した FS で構築されたパーティション上にファイルを配置する (図. 4). WAL ファイルは MemTable の容量制限とは別に制限が掛けられる. これは同じ key への書き換えなどの MemTable の容量に変化を与えない操作に対しても WAL の容量増加が発生するためである.

このため, WAL 自体にに容量の上限を設定する. WAL の割り当て容量を削減する場合には WAL の解放が必要出るため MemTable の設計同様に MemTable の書き出しを行う. PMDK では容量減少を RocksDB に要求したのち, MemTable が解放されたことを確認次第増加の必要な別プロセスに容量増加を通知する.

5.3 SSTable

SSTable は PM に対応した FS で構築されたパーティション上に Level 1 に対応するファイルのみを配置する (図. 5). RocksDB では各 Level 1 の総ファイル容量の上限を設定できるのでこれを利用して PM の使用量の上限を設定する.

また, この際に MemTable の容量をそのままにすると Level 1 の SSTable が一度の MemTable の書き出しのみで上限に達してしまう可能性があるため, 推奨される設定 をもとに MemTable の上限も変更する. PMDK では容量減少を RocksDB に要求したのち, SSTable が次の Level へ書き出されたことを確認次第増加の必要な別プロセスに容量増加を通知する.

6. 実験

本研究では設計に従い MemTable の割り当て容量変更機構について実装を行った. 他の対象及び PMDK については現在実装中である.

本研究では RocksDB に実装した MemTable の手法において PM の割り当て量を動的に変更した際にスループットが実際に変化することを確認する実験を行った. 実験では Intel(R) Xeon(R) Gold 6212U, SSD 500 GB, DRAM 96 GB および, Optane DC PM 128 GB を搭載したマシン上で計測した.

MemTable の容量を 128 MB - 1024 MB に設定した状態で 1M リクエストを実行したのち, 容量を 1024 MB に制限した状態で 1M リクエストを実行した. MemTable に対する実験では図. 6 に示す通り制限変更前は容量が大きいほどスループットが高くなることが確認され, 制限を行うことで直前の状態によらず 1024MB と同程度の性能に落ち着いた. 実験においては MemTable に対する割り当て容量の変化によって実際にスループットの変化を得ることが確認された. このことから, ある程度までは PM 容量を

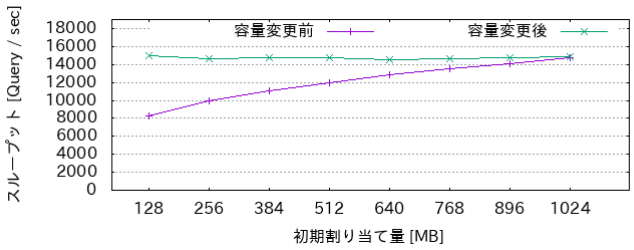

図 6 MemTalbe 割り当て容量変化前後のスループット

増加させることでスループットの向上がみられ, 負荷の小さい別プロセスから割り当てを切り替えることが実際に全体の性能に影響しうることが言える.

7. おわりに

本研究において DBMS が集約されたマシン上での PM 割り当てを調整することの考慮不足と, 実際に割り当て容量を変更することによる性能の変化を確認した. 既存研究においては PM が潤沢な環境を想定しているものと PM の制約があっても単一マシン上で単一の DBMS のみが稼働している環境を想定していた. 現実には PM は多数の用途での同時併用を想定したつくりとなっており, 以上のことから実際の用途の考慮が不十分であると言える.

本研究では MemTable の PM 利用を例に実際に PM 容量がその性能に影響することを実験において確認した. このことから PM の割り当てを考慮することの重要性があると言える. 今後は実際に設定されたポリシーに従いスループット等の評価値を最大化する割り当て容量調整機構について実施を行う予定である.

参考文献

[1] Intel optane technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.
[2] Intel®optane™technology. https://www.intel.com/optane.
[3] ipmctl. https://github.com/intel/ipmctl.
[4] libpmemobj. https://pmem.io/libpmemobj-cpp/.
[5] ndctl. https://github.com/pmem/ndctl.
[6] Persistent memory development kit. http://pmem.io/pmdk/.
[7] Rocksdb wiki. https://github.com/facebook/rocksdb/wiki.
[8] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: Diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.
[9] Apache. Hbase. https://hbase.apache.org/.
[10] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722, 2015.
[11] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. Nvmove: Helping programmers move to byte-based persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*, 2016.
[12] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, 13(12):2634–2648, 2020.
[13] Intel Corporation. Redis. https://github.com/pmem/redis/tree/3.2-nvml.
[14] Lenovo Corporation. Memcached. https://github.com/lenovo/memcached-pmem.
[15] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.
[16] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIXATC 20)*, pages 859–872, 2020.
[17] Ellis Giles, Kshitij Doshi, and Peter Varman. Hardware transactional persistent memory. In *Proceedings of the International Symposium on Memory Systems*, pages 190–205, 2018.
[18] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Software wear management for persistent memories. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 45–63, 2019.
[19] Google. Google cloud platform. https://cloud.google.com/why-google-cloud.
[20] Google. Leveldb. http://leveldb.org/.
[21] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. Rethinking logging, checkpoints, and recovery for high-performance storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 877–892, 2020.
[22] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482, 2017.
[23] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, 2018.
[24] Intel. Persistent collections for java. https://github.com/pmem/pcj.
[25] Intel and Micron. Intel and micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.
[26] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. Slm-db: single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.
[27] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Re-

ⓒ 2021 Information Processing Society of Japan

designing lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 993–1005, 2018.

[28] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[29] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krueger, and Martin Grund. High-performance transaction processing in sap hana. *IEEE Data Engineering Bulletin*, 36(2):28–33, 2013.

[30] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477, 2019.

[31] Jianhong Li, Andrew Pavlo, and Siying Dong. Nvmrocks: Rocksdb on non-volatile memory systems. `http://istcbigdata.org/index.php/nvmrocks-rocksdb-on-non-volatilememory-systems/`.

[32] Chen Luo and Michael J Carey. Efficient data ingestion and query processing for lsm-based storage systems. *arXiv preprint arXiv:1808.08896*, 2018.

[33] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. Agamotto: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064, 2020.

[34] Oracle. Java persistence api. `https://www.oracle.com/technetwork/jp/java/javaee/tech/persistence-jsp-140049.html`.

[35] Patrick O ' Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O ' Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[36] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276, 2014.

[37] Persistent Memory Programming. pmem-rocksdb. `https://github.com/pmem/pmem-rocksdb`.

[38] Persistent Memory Programming. pmem.io. `https://pmem.io/`.

[39] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[40] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 24–33, 2009.

[41] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[42] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. Programming for non-volatile main memory is hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–8, 2017.

[43] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 316–332, 2019.

[44] SNIA. Nvm programming model v1.2. `https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf`.

[45] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555, 2018.

[46] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. *ACM SIGOPS Operating Systems Review*, 28(5):86–97, 1994.

[47] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 70–83, 2018.

[48] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. Nvm-cached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 1–7, 2016.

[49] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–3, 2020.

[50] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–439, 2019.

[51] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496. ACM, 2017.

ⓒ 2021 Information Processing Society of Japan