

不揮発性メモリを使ったデータ永続化システム NV-HTM の 評価

飯干 寛幸^{1,a)} 松本 康太郎^{1,b)} 鵜川 始陽^{1,c)}

概要：近年、電源が失われても内容が保持されるメモリとして不揮発性メモリ (NVM) が市場に登場した。NVM を使ったシステムでは、データがキャッシュから NVM に書き出されて初めて永続化される。そのため、いつ電源が失われても良いように、常に NVM 上のデータの整合性を保つための方法が研究されている。その中には、個々のデータ構造を NVM に対応させるアプローチと、汎用のデータ構造を永続化するためのシステムを開発するアプローチがある。本研究では、NVM を使ってデータ構造を永続化するためのシステムである NV-HTM の評価を行った。そのために、NV-HTM を使い永続化した B⁺-木と、B⁺-木を NVM に特化させた FPTree の性能を比較した。NV-HTM は性能評価用の DRAM を用いたエミュレータを使うソースコードしか存在しなかったため、実機で動作するように修正が必要だった。FPTree もアルゴリズムの記述から実装する必要があった。これらを実装して比較したところ、NV-HTM は FPTree に比べて遅く、スレッド数を増やしても 4 スレッドまでしかスケールしなかった。そこでさらに、NV-HTM の性能ボトルネックを詳細に調査した。

キーワード：不揮発性メモリ、トランザクショナルメモリ、性能評価、B⁺-Tree

1. はじめに

近年、高速にランダムアクセス可能なストレージとして不揮発性メモリが注目されており、不揮発性メモリの実機も徐々に利用可能になっている [1]。不揮発性メモリは、DRAM と同じようにメモリバスに接続して、通常書き込みや読み込み命令を通じて利用することができる。これによって、高速かつ簡単にユーザプログラムによる書き込みを不揮発性メモリ上に残す (永続化) することができる。例えば、複数スレッドから同時にアクセスされることを前提として作られたデータ構造 (以下、並行データ構造と呼ぶ) を不揮発性メモリ上に置く。すると、計算機の電源が失われた後も残る、高速なデータ構造にすることができる。

ただし、不揮発性メモリへの書き込みはキャッシュを介して行われることに注意が必要である。キャッシュ上の情報は計算機から電源が失われると消えてしまうので、キャッシュに残っている書き込みは計算機から電源が失われると消えてしまう。したがって、キャッシュから不揮発性メモリへ書き戻されたデータのみが永続化され、それ以

外のデータは失われることになる。加えて、キャッシュが書き戻されるタイミングはユーザ側から把握できない。これによって、再び計算機を起動したときにはデータの一貫性が崩れた状態になってしまう。そのため、データの一貫性に関わるような書き込みについては、キャッシュを書き戻す命令を使って正しい順序で永続化されるようにする必要がある。並行プログラムで、自然に正しい順序で永続化されるような記述ができるように、トランザクショナルメモリを使う方法が提案されている [2], [3]。トランザクショナルメモリは、複数スレッドが同時に同じ変数へアクセスするのを防ぐための仕組みの一つである。これらの方法では、永続化されたデータにはトランザクション中でのみアクセスできる。以下では、これを永続的なトランザクショナルメモリと呼ぶ。永続的なトランザクショナルメモリは、コミット時に永続化を行う機能を持っている。永続化されたトランザクションは、計算機の再起動後に、その結果を復元することができる。NV-HTM [2] は、ハードウェアトランザクショナルメモリ (HTM) をベースとした永続的なトランザクショナルメモリの一つである。

一方、不揮発性メモリを使った永続化を前提として作られた並行データ構造 (以下、永続並行データ構造と呼ぶ) も開発されている [4], [5]。永続並行データ構造に比べて、永続的なトランザクショナルメモリを使って永続化した並

¹ 高知工科大学

a) iiboshi@pl.info.kochi-tech.ac.jp

b) kmatsumoto@pl.info.kochi-tech.ac.jp

c) ugawa.tomoharu@kochi-tech.ac.jp

行データ構造は、トランザクション全体を永続化するためのオーバーヘッドがかかることが予想される。

本研究では、不揮発性メモリの実機上における NV-HTM のパフォーマンスの評価を行う。そのために、NV-HTM を使い永続化した B⁺-木と、B⁺-木の永続並行データ構造である FPTree [4] を比較する。比較のための実験は不揮発性メモリを搭載した計算機上で行う。その後、二つの B⁺-木の性能差とその原因について考察する (6 章)。

そのために、まず公開されている NV-HTM のソースコードを実機上で動作するようにした (4.4 節)。また、不揮発性メモリ上に B⁺-木のデータを置くためのアロケータの実装も行なった (5 章 bptree)。

2. トランザクショナルメモリ

トランザクショナルメモリ [6] は、複数スレッドによる、同じ変数への同時アクセスを防ぐための仕組みの一つである。

2.1 トランザクション

トランザクションは、データベースにおいて用いられる概念で、連続する複数の処理をまとめた一単位である。トランザクション実行が完了することをコミットされる、失敗することをアボートされるという。トランザクション実行の結果はコミットされるかアボートされるかのいずれかである。トランザクションの処理は実行途中で他のトランザクションから観測できず、コミット後に処理結果が全て同時に観測できるようになる。二つのトランザクションが同じ対象を同時に処理する場合、少なくとも一つのトランザクションが対象に変更を加えていた場合、処理結果に不整合が生じることがある。このような場合をコンフリクトが起きたと表現する。コンフリクトが起きた場合、いずれかのトランザクションがアボートされる。トランザクションがアボートされたときは、トランザクション内で行われた変更を全て取り消し、トランザクション開始時点まで実行を巻き戻す。

2.2 トランザクショナルメモリの概要

トランザクショナルメモリは、並行プログラム中で同期の必要なメモリ操作をトランザクションとしてまとめて行う機能を提供する。トランザクションの処理対象はメモリになり、トランザクションを実行するのはスレッドになる。

2.3 HTM

Hardware Transactional Memory (HTM) はハードウェアによるトランザクショナルメモリの実装である。HTM は、ソフトウェアによるトランザクショナルメモリの実装に比べてオーバーヘッドが小さいという利点がある。一方で、特定の命令を実行した場合やプロセスが切り替わった

場合に起きるアボートなど、コンフリクト以外の原因によるアボートが発生する。

Restricted Transactional Memory [7] (RTM) は Intel による HTM の実装である。RTM では、トランザクションのアボートが発生した際に、原因を示す値がレジスタに設定される。コンフリクトが起きた場合の他に、トランザクション内におけるメモリアクセス数の上限を超えた場合にもアボートが発生する。プログラムから明示的にアボートさせることもできる。その他、トランザクション内でキャッシュラインを書き戻す命令を実行した場合もアボートするが、アボートの原因はその他に分類される。


3. 不揮発性メモリの使用方法

不揮発性メモリを専用のファイルシステムで mmap することで、バッファを介さずに直接不揮発性メモリにアクセスできる。ただし、不揮発性メモリのアクセスはキャッシュを介して行われる。キャッシュラインを書き戻す命令によって書き込みを不揮発性メモリに反映させ、永続化することができる。

Intel のプロセッサでは、キャッシュラインを書き戻す命令として、一般的なキャッシュラインを書き戻す命令である `clflush` に加えて新しく `clwb` 命令が提供されている [8]。 `clflush` 命令では、キャッシュラインを書き戻すときに対象のキャッシュラインを捨ててしまう。 `clflush` 命令を発行した後も引き続き同じアドレスにアクセスする場合には、再びキャッシュに読み込むための時間が無駄にかかる。 `clwb` 命令は `clflush` より最適化されたキャッシュライン書き戻し命令で、キャッシュラインを捨てずに書き戻す、依存がない他の書き込み命令を先に実行できるという特徴がある。不揮発性メモリを使ったシステムでは、データの一貫性を保つために、書き込みを永続化する順番が重要なことがある。そのときは、 `clwb` 命令だけではキャッシュラインの書き戻しが保証されず、 `clwb` 命令の実行が完了するまで後続の書き込み命令の実行を待つメモリバリア命令を使う必要がある。ただし、メモリバリア命令は時間がかかる命令なので、パフォーマンスのためには使用を最小限にとどめる必要がある。

4. NV-HTM

4.1 NV-HTM の概要

NV-HTM は、HTM のトランザクションの結果を永続化するためのシステムである。NV-HTM の全体図を  1 に示す。NV-HTM では、不揮発性メモリ上に Persistent Snapshot と呼ばれるユーザデータ領域とログ領域の二つを確保する。ユーザのプロセス (以下、Worker プロセス) は、Persistent Snapshot に書き込む代わりに、DRAM 上に Copy-on-Write で作られる Persistent Snapshot のコピーに書き込む。このコピーは Working Snapshot と呼ばれる。

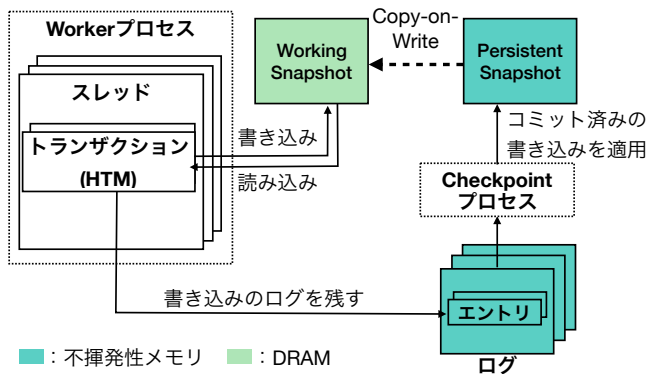


図 1 NV-HTM の全体図 ([2] の Fig. 1 を基に作成)

Worker プロセスは、HTM のトランザクションの中でのみ Working Snapshot への書き込みを行う。HTM のトランザクション中では、Working Snapshot へ書き込むたびに、書き込み内容を記録したログエントリを不揮発性メモリ上のログに追加する。トランザクションの実行が完了すると、コミットが完了したことを示すログエントリをログに追加する。Persistent Snapshot には、Checkpoint プロセスと呼ばれるプロセスが書き込む。Checkpoint プロセスは、コミットが完了したトランザクションのログの内容を Persistent Snapshot に反映する。その後、Checkpoint プロセスは反映済みのログエントリを削除する。次の計算機の起動時にログが残っていると、Persistent Snapshot に反映されていないトランザクションがある可能性がある。そのときは、Checkpoint プロセスを起動して、Persistent Snapshot にログを反映させる。これにより、コミットが完了したトランザクションが全て実行された後の状態を復元することができる。

4.2 トランザクション実行

NV-HTM のトランザクションは、HTM を使ったトランザクション実行とログの永続化で構成される。ログの永続化が完了したとき、NV-HTM のトランザクションがコミットされる。NV-HTM のトランザクションのコミットと、HTM を使ったトランザクションのコミットと区別するため、前者を永続コミット、後者を非永続コミットと呼ぶ。なお、Intel プロセッサを使う場合は、HTM の実装として RTM を使用する。NV-HTM のトランザクション実行の流れを図 2 に示す。HTM を使ったトランザクション実行では、Worker プロセスは Working Snapshot に対する書き込みとログの追加を行う。HTM を使ったトランザクション実行中にアボートを起こさず、非永続コミットできた場合はログの永続化に移る。HTM を使ったトランザクションが一定回数アボートした場合は、グローバルなロックを取得して他プロセスをブロックしながらトランザクションを実行する。

ログの永続化は 2 段階で行われる。まず、HTM を使っ

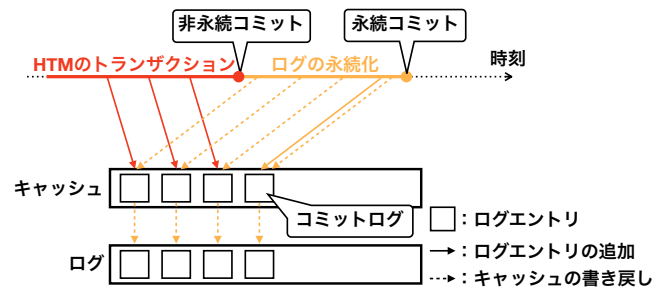


図 2 NV-HTM のトランザクション実行

たトランザクション実行中に追加されたログ全てに対してキャッシュラインを書き戻す命令を発行する。それから、コミットが完了したことを示すログエントリ（以下、コミットログと呼ぶ）をログに追加し、コミットログに対してキャッシュラインを書き戻す命令を発行する。コミットログが永続化されたとき、ログの永続化が完了する。

4.3 Checkpoint プロセス

Checkpoint プロセスは、プログラム開始時に fork して生成され、待ち状態になる。その後、Worker プロセスからの要求によって動作する。

Worker プロセスは、ログの容量が不足したときに Checkpoint プロセスを動作させる。Worker プロセスが Checkpoint プロセスを動作させる条件を図 3 に示す。Checkpoint プロセスを動作させるタイミングは三つあり、

- (1) トランザクションの開始時
- (2) トランザクション中の書き込み時
- (3) ログの空き容量不足によるトランザクションのアボート後

である。

(1) のトランザクションの開始時に、ログの空き容量がログ全体の半分を切っていた場合、Checkpoint プロセスを動作させてからトランザクションを開始する。このとき Worker プロセスは Checkpoint プロセスの終了を待たない。これは、ログに空きがある段階で Checkpoint プロセスが動作することで、Worker プロセスがログ容量不足で待ち状態になる前にログを処理することを目的としている。

(2) のトランザクション中の書き込み時に、ログの空き容量が不足していた場合、HTM を使った実行中ならば Checkpoint プロセスを動作させずアボートを起こす。グローバルなロックを使った方法に移行していた場合は、16 エントリ分の空きがなければ Checkpoint プロセスを動作させる。Worker プロセスは、その後 32 エントリ分の空きができるのを待ってから実行を再開する。これは、トランザクションの続行に必要なログ容量を確保することを目的としている。

- (3) のログの空き容量不足によるトランザクションの

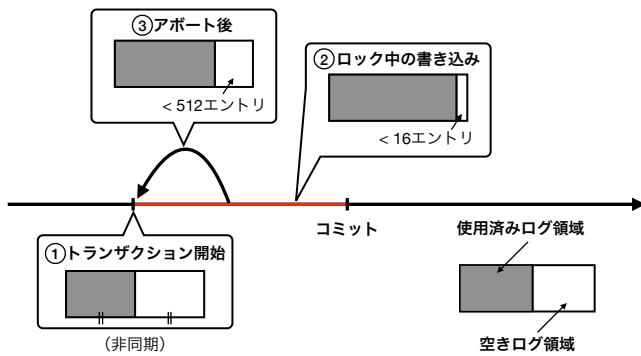


図 3 Checkpoint プロセスの開始条件

アボートが起きた後、512 エントリ分の空きがなければ Checkpoint プロセスを動作させ、512 エントリ分の空きができてから再度トランザクションを実行する。これは、ログ容量不足によるアボートを繰り返さないことを目的としている。

Checkpoint プロセスでは、ログの処理にあたって次の二つの効率化の仕組みが導入されている。まず、新しいトランザクションから順に処理する。複数のトランザクションにおいて同じアドレスに対する書き込みがあった場合、その中で最新のトランザクションによる書き込みだけを反映する。古いトランザクションによる書き込みは、結局新しいトランザクションによる書き込みで上書きされるからである。これにより、同じアドレスに対する書き込みを省略でき、不揮発性メモリに対する書き込み量を減らせる可能性がある。複数トランザクションによる同じアドレスへの書き込みを検出するために、Checkpoint プロセスは書き込みを行うと同時に書き込まれたアドレスを記録する。記録済みのアドレスへの書き込みは古い書き込みとして無視する。

次に、キャッシュを書き戻す命令の発行は最後に行なっている。Persistent Snapshot への書き込みを永続化するために、Checkpoint プロセスはキャッシュラインを書き戻す命令を発行する。このとき、ログのエントリを一つ反映するたびにキャッシュラインを書き戻す命令を発行するのではなく、一通りログのエントリを反映し終えてから発行する。これにより、同じキャッシュラインに対する書き込みがあった場合、キャッシュラインを書き戻す命令をまとめることができる。同じキャッシュラインへの書き込みを検出するために、Checkpoint プロセスは一つ目の仕組みと同様に書き込みのあったアドレスを記録する。

4.4 NVM の使用

NV-HTM のソースコードは公開されている^{*1}。しかし、この実装は不揮発性メモリを使わず、不揮発性メモリのエミュレータを使用するものである。エミュレータは、不揮

*1 https://bitbucket.org/daniel_castro1993/nvhtm/src/master/

発性メモリに対する書き込みのオーバーヘッドを再現する。エミュレータでは、本来不揮発性メモリに置くデータを DRAM 上に置く。その上で、エミュレータ上のデータに対するキャッシュラインを書き戻す命令には決まった時間のディレイを挿入する。公開されているソースコードから、不揮発性メモリを使用するように変更した箇所は次の通りである。

- (1) 不揮発メモリ上にデータを置くように変更
- (2) キャッシュライン書き戻し命令を `clwb` に変更し、必要に応じてメモリバリア命令を挿入

実際の不揮発性メモリを使って実験するため、不揮発性メモリ上に作成したファイルをメモリマップして使用するよう変更した。(2) については、ディレイの挿入になっていたキャッシュライン書き戻し命令を、最適化されたキャッシュライン書き戻し命令である `clwb` 命令を使用するよう変更した。メモリバリア命令は二箇所に入れた。一つ目は、Checkpoint プロセスが Persistent Snapshot への書き込みに対するキャッシュライン書き戻し命令を全て発行した直後である。Checkpoint プロセスは、ログの内容を Persistent Snapshot に反映し終わると、反映が済んだログエントリを削除する。このとき、Persistent Snapshot への書き込みが永続化されてからログを削除するためである。二つ目は、Worker プロセスがコミットログをログに追加する直前である。書き込み内容を記録したログが全て永続化されてから、コミットログを追加するためである。

5. B⁺-木とその永続化

本研究では、NV-HTM を使って永続化した B⁺-木の性能を測ることで NV-HTM の性能を調べる。本章では、そのために本研究で実装した B⁺-木と、比較対象である FPTree の実装について述べる。

実装した B⁺-木は、各種操作を NV-HTM のトランザクションとすることで永続化した。初期化処理を含め、不揮発性メモリに対する書き込みは全て NV-HTM を通して行なっている。

5.1 B⁺-木

B⁺-木 [9] は平衡多分木である。内部ノードにはキーのみを持ち、葉ノードはキーと値を持つ。葉ノードは連結リストになっており、高速なシーケンシャルアクセスが可能である。内部ノードと葉ノードのキーは全て昇順にソートされている。これによって、高速な二分探索を使ってキーを探すことができるようになる。B⁺-木の内部ノードがもつ子ノードは次のような条件を満たす。

子ノードが内部ノードの場合、親ノードのあるキー x_n より左側にある子ノードがもつキーは全

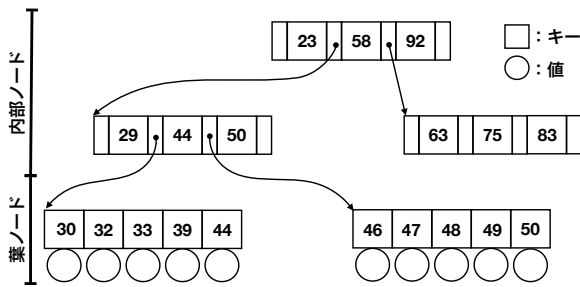


図 4 B⁺-木のノード例

て x_n より小さく, x_n より右側にある子ノードがもつキーは全て x_n より大きい. 子ノードが葉ノードの場合, 親ノードのあるキー x'_n より左側にある子ノードがもつキーは全て x'_n 以下であり, x'_n より右側にある子ノードがもつキーは全て x'_n より大きい.

条件を満たすノードの例を図 4 に示す. 最上段の内部ノードは子ノードとして内部ノードをもつ. 最上段の内部ノードがもつキー 58 の, 左側にある子ノードは 23 より大きく 58 より小さいキーをもち, 右側にある子ノードは 58 より大きく 92 より小さいキーをもつ. 中段の内部ノードは子ノードとして葉ノードをもつ. 中段の内部ノードがもつキー 44 の, 左側にある子ノードは 29 より大きく 44 以下のキーをもち, 右側にある子ノードは 44 より大きく 50 以下のキーをもつ.

B⁺-木に対して可能な操作には挿入, 削除, 検索がある. 挿入操作は, 新しいキーと値を葉ノードに追加する操作である. 一つの葉ノードがもつ要素数の上限を超える場合は, 葉ノードを分割する. 葉ノードは, 分割対象となる葉ノードの要素を半分だけ新しい葉ノードに移すことで分割する. まず, 分割対象となる葉ノードの中央にあるキーを親ノードに挿入し, そのキーより大きいキーを新しいノードに移す. そして, 新たに親ノードに挿入したキーの左側の子を分割した葉ノード, 右側の子を新しい葉ノードとする. 親ノードが満杯になった場合は, 親ノードを分割する. 内部ノードの分割は, 親ノードに挿入したキーを分割対象のノードから削除する点を除けば葉ノードと同じである. 削除操作は, 削除対象として指定されたキーが B⁺-木内のいずれかの葉ノードに存在するならば, そのキーを持つ要素を削除する操作である. 葉ノード内で削除対象の要素より右にある要素を一つずつ左にずらすことで削除する. 要素が最後の一つにだった場合は, 葉ノードを削除し, 親ノードから削除した葉ノードへのポインタとその右側 (無ければ左側) のキーを削除する. 検索操作は, 検索対象として指定されたキーを持つ要素を B⁺-木から探し出す操作である. B⁺-木の根ノードから順に, B⁺-木のノードが満たす条件に基づいて検索対象のキーを持つ葉ノードを探す.

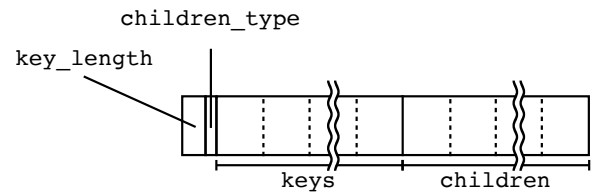


図 5 B⁺-木の内部ノード

5.2 NV-HTM を使った B⁺-木永続化

本研究では文献 [9], [10] に示された B⁺-木のアルゴリズムを実装した. さらに, B⁺-木のノードを不揮発性メモリ上に置くために, 不揮発性メモリの領域を管理するメモリアロケータを作成した. 本節では, B⁺-木のノードの構成とメモリアロケータ, NV-HTM を使った永続化方法について述べる. なお本研究の B⁺-木では, 5.3.1 節で後述する FPTree [4] と同じ理由で葉ノードのみを永続化する.

5.2.1 ノードの構成

内部ノードの構成を図 5 に示す. `key_length` は 4 バイトで有効なキーの数, `children_type` は子ノードの種類を表す 1 バイトのフラグ, `keys` は各 8 バイトのキーの配列, `children` は子ノードへのポインタの配列で, 各 8 バイトである. 最大キー数は 128 である. 本研究の実装では, 有効なキーは左づめになるように保っている. これによって `key_length` を使ってキーが有効なものか判断できるようになる. さらに, 内部ノードのキーは常に昇順にソートされた状態になるように, 挿入ソートのアルゴリズムにしたがって挿入する.

内部ノードには子ノードとして内部ノードをもつものと葉ノードをもつものがある. `children` はノードの種類に関する情報を持たないため, 子ノードとして内部ノードと葉ノードのどちらをもつかを `children_type` に入れる値で区別する.

葉ノードの構成を図 6 に示す. `tid` はノードを作成したスレッドを表す 1 バイトの ID, `next` は次の葉ノードへのポインタ, `prev` は前の葉ノードへのポインタ, `pnext` は 8 バイトのファイル ID と 8 バイトのオフセットを組にした次のノードへの永続ポインタ [4], `key_length` は 4 バイトの有効なキーの数, 各 12 バイトの要素 (キーと値の組) `kv` からなる. 要素の最大数は 128 である. `tid` はのちに述べるメモリアロケータで必要な値である. `next`, `prev` は葉ノードにシーケンシャルアクセスするためのものである. また, `pnext` は, 計算機の再起動後に不揮発性メモリが `mmap` されるアドレスが変わってもよいように, 永続ポインタになっている.

5.2.2 再起動後の復元

`pnext` によって, 葉ノードは連結リストになっている. そのため, 再起動後でもリスト先頭の葉ノードから全ての葉ノードを見つけ出すことができる. 葉ノードから内部ノードを再構成することで, B⁺-木を復元できる. 復元は

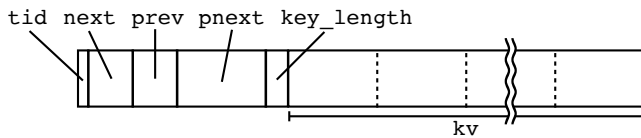


図 6 B⁺-木の葉ノード

空の B⁺-木に対して、pnext をたどりながら葉ノードを全て挿入することで行う。

5.2.3 メモリアロケータ

B⁺-木を作成するにあたって、不揮発性メモリの領域を管理するためのメモリアロケータを作成した。本研究の B⁺-木は並行化して使うことを前提としている。そのため、メモリアロケータは競合を起こしにくいように設計する必要がある。なお、上述したように、今回は葉ノードのみ永続化するため、一回のメモリ割り当てで割り当てられる領域は全て同じ大きさである。そこで、本研究のメモリアロケータでは、割り当てできる領域の大きさを一種類に制限する。また、葉ノードが連結リストになっているため、計算機の再起動が起きても割り当て済みの領域は葉ノードのリストを辿ること確認できる。そのため、メモリアロケータの管理情報は全て DRAM 上に置く。

メモリアロケータの全体を図 7 に示す。作成したメモリアロケータは、グローバルなメモリプールと、スレッドローカルな allocation buffer (TLAB) で構成される。TLAB は並行にメモリ割り当てを行なった際の同期コストを減らすことを目的としている。

TLAB は、各スレッドが持つ全スレッド分のキューによって管理される。各スレッドは自身の ID に対応したキューにある空き領域からメモリを割り当てることができる。例えば、図 7 のスレッド A は、スレッド A の持つスレッド A 用のキューとスレッド B の持つスレッド A 用のキューにある空き領域の両方からメモリを割り当てることができる。所持者と使用者が一致しているキューには、ロックを使用せずアクセスすることができる。そのため、あるスレッド A が新たな領域を必要とする場合、スレッド A が持つスレッド A 用のキューを確認する (図 7 の①)。そこに空き領域がなかった場合は、他スレッドが持つ自身の ID に対応したキューを確認する (図 7 の②)。このとき、空き領域を追加するスレッドとの競合を避けるため、対象のキューをロックしてからメモリ割り当てを行う。そこにも空き領域がなかった場合は、グローバルな空き領域を確認する (図 7 の③)。このとき、他スレッドとの競合を避けるため、ロックを使って排他制御しながらメモリ割り当てを行う。その後、割り当てた領域にはスレッド ID を書き込んでおく。領域の解放は次のように行う。まず、解放する領域に書き込まれているスレッド ID を確認する。そして、領域を解放するスレッドが持つ、解放する領域に書き込まれているスレッド ID に対応したキューに追加する。

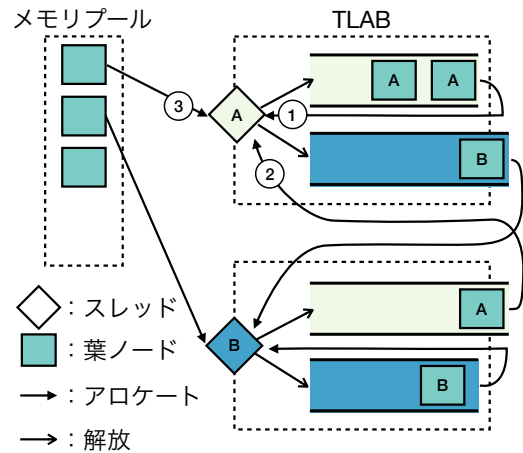


図 7 アロケータの全体図

このとき、領域を解放するスレッドの ID と解放する領域に書き込まれているスレッド ID が異なる場合は、キューをロックしてから追加する。したがって、一度割り当てられた領域はグローバルな空き領域には戻らない。

5.3 FPTree

FPTree [4] は、不揮発性メモリに特化した B⁺-木の一種である。FPTree でも B⁺-木に対する操作は HTM を使ったトランザクションとして実現されている。

不揮発性メモリに対する書き込み・読み込みは、高速ではあるが DRAM に比べれば遅い。そのため、不揮発性メモリに対する書き込み・読み込み回数が多いとパフォーマンスが落ちてしまう。加えて、HTM ではトランザクション中でキャッシュを書き戻す命令を発行するとトランザクションがアバートされてしまう。そのため、HTM を使った並行化と永続化を両立させるためには工夫が必要になる。以下では、これらの問題を FPTree がどのように解決しているかを説明する。

5.3.1 永続化の対象

FPTree では、葉ノードは不揮発性メモリ上へ置き、再構成可能な内部ノードは DRAM 上へ置くという方針をとっている。読み込みが大量に発生する内部ノードについては DRAM 上に置くことで、不揮発性メモリへのアクセスを減らして木に対する操作を高速化できる。ただし、これによってデータ構造を回復する際の時間はより多くかかるようになる。

5.3.2 葉ノード

FPTree の葉ノードの構成を図 8 に示す。bitmap は要素の有効・無効を表すビットマップ、next は次の葉ノードへの永続ポインタ、fingerprint は各要素のキーごとに求めた 1 バイトのハッシュ値、kv はキーと値の組でできた要素、lock は葉ノードのロック状況を表す値である。FPTree では、葉ノードに要素をソートせずに格納する。また、要素が削除されたときに隙間を詰めることもしない。

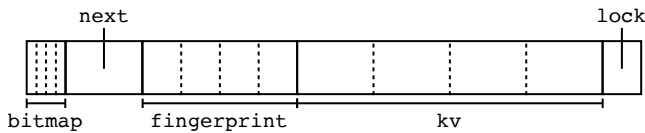


図 8 FPTree の葉ノード ([4] の Figure 2 を基に作成)

要素が有効であるか否かは葉ノードのヘッダにあるビットマップによって管理される。これは不揮発性メモリへのアクセスを減らすためである [11]。もし、ソートされた状態を保とうとすると、挿入と削除の両方で大量の要素の移動が発生する。

葉ノードの要素がソートされていないので、葉ノード内の要素を検索するときは線形探索によることとなる。線形探索では、不揮発性メモリからの読み込みが多く発生する。不揮発性メモリからの読み込みを減らすために、葉ノード内にある各要素について、1バイトのハッシュ値を葉ノードのヘッダに格納する。このハッシュ値は fingerprint と呼ばれる。fingerprint は 1 バイトなので、最大で一度の読み込みにより 1 キャッシュライン分の fingerprint を同時にキャッシュへ読み込むことができる。キーを比較する前に fingerprint を比較することで、比較のためにキーの値を不揮発性メモリから読み込む回数を減らすことができる。

5.3.3 HTM とロックの併用

HTM の中ではキャッシュラインを書き戻す命令を発行できない。HTM を使った並行化と永続化を両立するために、FPTree では HTM とロックを併用している。DRAM 上にある内部ノードの変更と、変更を伴わない葉ノードからの読み込み、葉ノードのロックには HTM を使う。一方、不揮発性メモリ上に存在する葉ノードの変更にはロックを使う。FPTree の操作は次のような流れになる。

1. HTM のトランザクション内で操作対象の葉ノードを探す。
2. 同トランザクション内で葉ノードをロックする。
3. 葉ノードに対して操作を行う。
4. 葉ノードをアンロックする。
5. 親ノードの更新が必要な場合は、新たに HTM のトランザクションを開始し更新する。

葉ノードのロックとアンロックは、単に葉ノードの lock に 1 または 0 を書き込むことで行う。これは、葉ノードのロックを HTM 内で行うことで、アトミック命令を使用する必要がなくなるためである。

5.3.4 葉ノード分割の実装

葉ノードを分割するとき、分割対象のノードには中央のキー以下のキーを、新しいノードには中央のキーより大きいキーを入れる必要がある。FPTree の葉ノードは要素を

ソートしないため、このときキーの大きさ順を調べる必要がある。この点は文献 [4] に記述がなかったため、分割するとき DRAM 上にインデックスを作成して、インデックスをソートするように実装した。葉ノードの要素を実際にソートすると、不揮発性メモリへの書き込みが多く発生するためこのような実装としている。

6. 実験

本章では、NV-HTM のパフォーマンス評価とその結果につながった要因を調べるために行なった実験について述べる。まず、NV-HTM を使って永続化した B⁺-木 (以下、B⁺-Tree_{NH}) のパフォーマンスを調べる。比較対象として、FPTree に加えて次の二つの B⁺-木を実装した。一つは、NV-HTM のオーバーヘッドがない場合として、木全体が DRAM 上にあり、HTM を使用する B⁺-木 (以下、B⁺-Tree_C) である。もう一つは、不揮発性メモリへのアクセスにかかるオーバーヘッドがない場合として、ログや Persistent Snapshot を DRAM 上に配置した NV-HTM を使う B⁺-Tree_{NH} (以下、B⁺-Tree_{NH}(DRAM)) である。B⁺-Tree_{NH} でも、キャッシュラインの書き戻し命令やメモリバリア命令は、通常の NV-HTM と同様に発行する。いずれの実験でも最初に 2,500,000 個の要素を挿入した木を作成した後、2,500,000 回の操作を行っている。スレッド数によらず、全体で 2,500,000 回実行されるように各スレッドに操作回数を均等に割り当てている。

6.1 実験設定

実験を行なった環境を示す。

- CPU: Intel Xeon Gold 6240 (2.60GHz, 18 コア, キャッシュラインサイズ 64B)
- メモリ: Micron Technology DDR4 48GB
- 不揮発性メモリ: Intel OPTANE DC persistent memory (256GB)
- OS: Ubuntu 18.04.3 LTS
- C コンパイラ: GCC version 7.4.0
- 不揮発性メモリのファイルシステム: NOVA filesystem [12]

6.2 実行時間とスケール性能の測定

B⁺-Tree_{NH} の実行時間とスケール性能を調べるために、スレッド数が増えたときの、B⁺-Tree の各種操作にかかる実行時間を測定した。図 9 はその結果である。なお、NV-HTM のログ容量は 1 スレッド当たり 40MiB としている。また、NV-HTM のログ処理による実行時間への影響を調べるために、Worker プロセス終了後に Checkpoint プロセスが再び待ち状態になるまでの時間を含めている。

B⁺-Tree_{NH} では、検索操作ではスレッド数が増えると実行時間が短くなっている一方、挿入操作と削除操作では

それぞれ4スレッドと8スレッドを超えると実行時間が長くなっている。このことから、 B^+ -Tree_{NH} は書き込み命令のある操作ではほとんどスケールしていないことがわかる。なお、検索操作が異なる傾向を示すのは、NV-HTMが読み込み操作においてログを残さないからだと考えられる。また、FPTreeと B^+ -Tree_{NH}を比較すると、検索操作を除けばスレッド数が増えるほど実行時間の差が大きくなっている。最も差が小さい1スレッドのときでも挿入操作と削除操作でそれぞれ5倍と6倍になっており、大きく差が開いている。したがって、FPTreeに比べて B^+ -Tree_{NH}はスケール性能も悪く遅い。さらに、挿入操作と削除操作では、 B^+ -Tree_{NH}(DRAM)と B^+ -Tree_{NH}の間にも実行時間に差がある。このことから、不揮発性メモリにアクセスするときのオーバーヘッドが B^+ -Tree_{NH}の実行時間に影響を及ぼしていることがわかる。加えて、 B^+ -Tree_{NH}(DRAM)と B^+ -Tree_Cを比較すると、特に挿入操作と削除操作において B^+ -Tree_{NH}(DRAM)の方が実行時間が長く、スケール性能も悪い。これは、NV-HTMを使うことによるオーバーヘッドが大きく表れていると考えられる。

6.3 B^+ -Tree_{NH}のボトルネックの調査

以下では、挿入操作と削除操作を行うときに B^+ -Tree_{NH}が遅くなっている原因を調べるための実験について述べる。 B^+ -Tree_{NH}の遅くなる原因として次の三つの可能性が考えられる。まず、NV-HTMを使うことでHTMのトランザクションが多くアボートされている可能性である。NV-HTMは本来の書き込み命令に加えてログに対する書き込み命令を追加で必要とする。これにより、HTMのトランザクションを実行する時間が長くなり競合が起きやすくなるほか、HTMのトランザクション内におけるメモリアクセス数の上限を超える可能性があるためである。

次に、Checkpointプロセスによるログ処理を待つ時間が長くなっている可能性である。ログ処理の待ち時間は次の二種類である。一つは、Workerプロセスの実行中にログ容量が不足したことによる待ち時間である。このときWorkerプロセスはログが処理されて空きができるまで待機する。もう一つは、Workerプロセスの終了後に残ったログを処理するための時間である。

最後に、不揮発性メモリへのアクセス時間が多くかかっている可能性である。これら三つに分けて調査を行った。

6.3.1 HTMのトランザクションのアボート

まず、アボートが起きたことによる実行時間への影響を調べるため、Workerプロセスが本来の処理以外に消費した時間を測定した。図10は、次の三つの原因によってWorkerプロセスが消費した時間を、原因毎に合計してスレッド数で割ったものである。Abortは、アボートが発生したトランザクションの、トランザクション開始からア

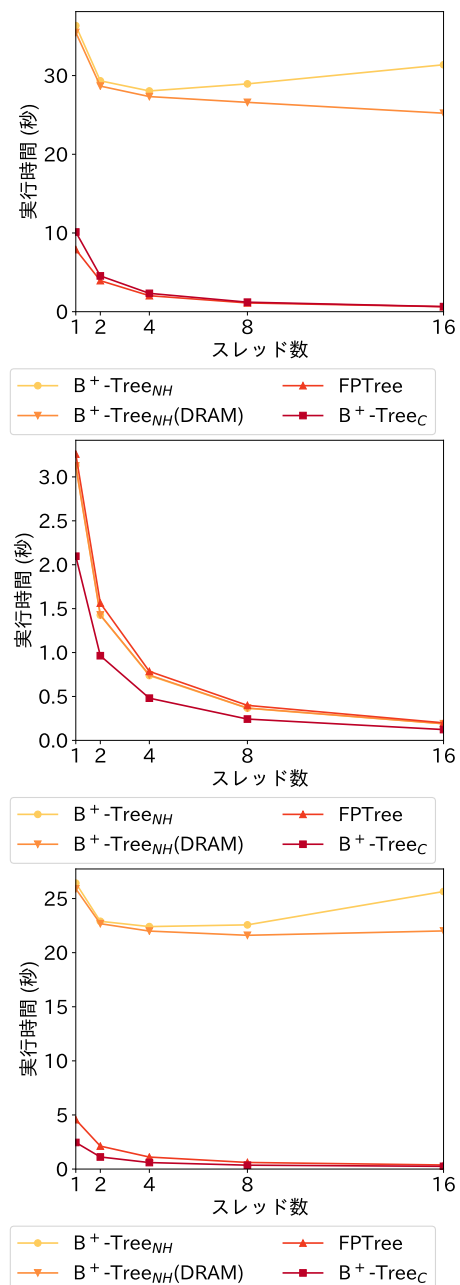


図9 スケール性能測定実験の結果(上:挿入操作, 中:検索操作, 下:削除操作)

ボートが起きるまでの時間である。Checkpoint-blockは、ログ容量が不足したためにCheckpointプロセスによるログの処理が終わるまで待機した時間である。HTM-blockは、あるスレッドがアボート回数の上限を超え、ロックを使った実行に切り替わったとき、その他のスレッドがトランザクションを実行できずに待機した時間である。

Abortに着目すると、消費された時間はスレッド数が1のときに最も長く、スレッド数が増えるほど短くなっている。特に、スレッド数が1のときには全体の半分を占めている。このことから、スレッド数が少ないときにはHTMのトランザクションのアボートがNV-HTMの長い実行時間の一因になっていること、スレッド数が多いときには

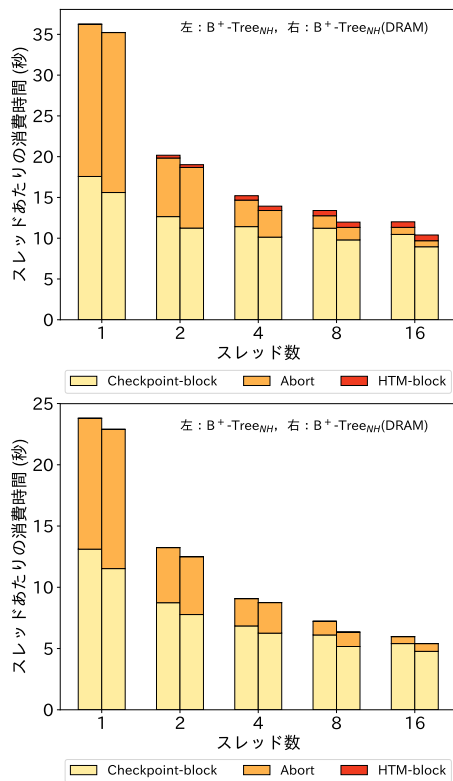


図 10 Worker プロセスが処理を行えなかった時間 (上:挿入操作, 下:削除操作)

HTM のトランザクションのアボートは実行時間にほとんど影響していないことがわかる。

そこで、HTM のトランザクションがアボートされる原因を調べるために、原因別のアボート回数を数えた。図 11 は、アボートの原因ごとに分けて集計したアボート回数のグラフである。conflict は競合が起きた場合、capacity はトランザクション内におけるメモリアクセス数の上限を超えた場合、explicit はログ容量が不足した場合、other は原因不明である。スレッド数が 1 のときには原因不明のアボートが最も多く、スレッド数が増えると徐々に競合によるアボートが増えている。図 10 ではスレッド数が増えるとアボートにより消費される時間が減っていた。したがって、conflict によるアボートはそれほど時間を消費せず、other によるアボートが最も多くの時間を消費していると考えられる。この傾向は削除操作において特に顕著である。other によるアボートを引き起こしている要因については特定できなかった。

6.3.2 ログ処理の完了待ち

Checkpoint プロセスが行うログ処理を待つ原因には、ログの残量不足による待ちと、Worker プロセス終了後に残りのログを処理するための待ちの二つがある。前者で消費された時間は、図 10 における Checkpoint-block が相当する。後者で消費された時間を図 12 に示す。前者ではスレッド数が増加するにしたがって待ち時間が減少している一方で、後者ではスレッド数が増加するにしたがって待ち時間

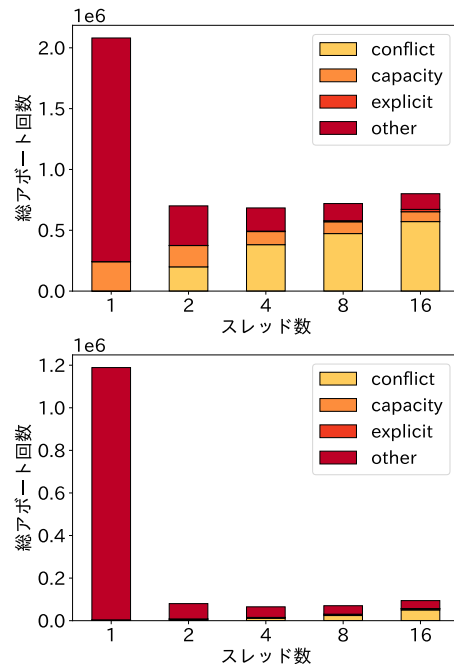


図 11 原因別アボート回数 (上:挿入操作, 下:削除操作)

が増加している。前者における待ち時間の減少は、スレッド数が増えることによるログ容量の増加が原因だと考えられる。スレッドあたりのログ容量は一定なので、Worker プロセス全体でのログ容量はスレッド数が多いほど大きくなる。これによって、Worker プロセスと Checkpoint プロセスが並行に動作する割合が大きくなったと考えられる。後者における待ち時間の増加は、Checkpoint プロセスの処理がシングルスレッドで行われていることが原因だと考えられる。Checkpoint プロセスはシングルスレッドのままなので、同量のログを処理するために必要な時間はあまり変わらない。一方で、前者における待ち時間の減少やアボートによる時間消費の減少によって Worker プロセスは処理を早く終わるようになる。よって、後者による待ち時間が増えたと考えられる。

6.3.3 不揮発性メモリへのアクセス

不揮発性メモリへのアクセス時間による Checkpoint プロセスへの影響を調べるために、 B^+ -Tree_{NH}(DRAM) でも Checkpoint プロセスが行うログ処理を待つ時間を測定した。図 10 の Checkpoint-block に示す、 B^+ -Tree_{NH}(DRAM) のログの残量不足による待ち時間は、 B^+ -Tree_{NH}のものに比べて挿入操作で約 86%から 89%、削除操作で約 85%から 91%だった。このことから、不揮発性メモリへのアクセス時間によって Checkpoint プロセスがログを処理するために必要な時間が長くなっていると考えられる。一方で、図 12 に示す、 B^+ -Tree_{NH}(DRAM) の Worker プロセス終了後に残りのログを処理するための待ち時間を見ると、スレッド数が 2 から 8 のとき B^+ -Tree_{NH} に比べて挿入操作では約 5%から 7%、削除操作で約 2%から 5%長くなっている。この待ち時間の増加の原因は不明である。

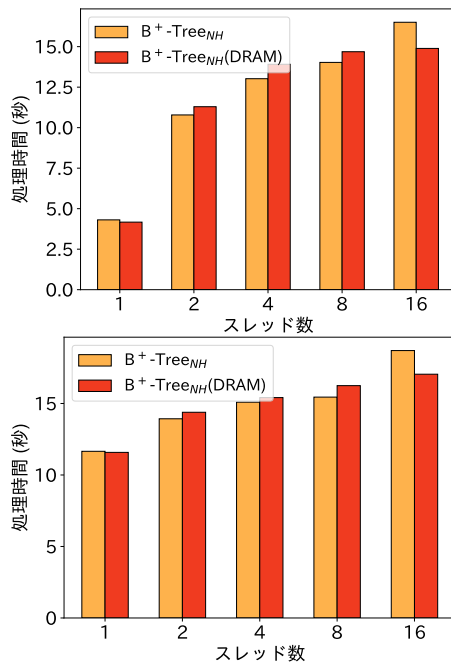


図 12 Worker プロセス終了後のログ処理時間 (上:挿入操作, 下:削除操作)

7. 関連研究

NV-HTM が提案された文献 [2] でも NV-HTM パフォーマンス評価は行われている。しかし、実験は不揮発性メモリの代わりに DRAM を用いたエミュレータ上で行われており、また比較対象は別の永続的なトランザクショナルメモリである PHTM[3] であった。本研究では、不揮発性メモリの実機上で、NV-HTM を使った B⁺-木と B⁺-木の永続並行データ構造との比較を行なった。

また、FPTree が提案された文献 [4] では FPTree 以外の B⁺-木の永続並行データ構造や一般的な B⁺-木との比較を行なっている。しかし、こちらも DRAM を用いたエミュレータ上で実験しており、永続的なトランザクショナルメモリを使った B⁺-木との比較は行われていない。文献 [5], [11], [13] も B⁺-木の永続データ構造を提案しており、同様にエミュレータ上で実験している。

文献 [14] では、不揮発性メモリの実機を使用し、不揮発性メモリのパフォーマンスを様々な角度から測定している。Key-Value ストアに組み込まれた形で B⁺-木のパフォーマンス測定も行われているが、永続的なトランザクショナルメモリを使った B⁺-木との比較は行われていない。

8. おわりに

本研究では、永続的なトランザクショナルメモリの一つである NV-HTM の、不揮発性メモリを搭載した計算機上でのパフォーマンスを評価した。そのために、NV-HTM を使って永続化した B⁺-木を実装し、また比較対象として永続並行データ構造である FPTree を実装した。実装し

た各種の B⁺-木のスケール性能を調べた結果、NV-HTM を使って永続化した B⁺-木は FPTree に比べて遅く、4 スレッドまでしかスケールしなかった。NV-HTM を使って永続化した B⁺-木が遅い原因を調べる実験をさらに行った結果、NV-HTM のログ処理速度が大きな要因となっており、スレッド数が少ないときには原因不明のアボートにより消費される時間も多くあることがわかった。NV-HTM のログ処理速度が遅い原因は不明である。

参考文献

- [1] インテル: インテル®Optane™DC パーシステント・メモリー, Intel Corporation (オンライン), 入手先 (<https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/optane-dc-persistent-memory.html>) (参照 2020-01-05).
- [2] Daniel, C., Paolo, R. and Barreto, J.: Hardware Transactional Memory Meets Memory Persistency, *Proc. IPDPS'18*, pp. 368–377 (2018).
- [3] Avni, H., Levy, E. and Mendelson, A.: Hardware Transactions in Nonvolatile Memory, *Proc. DISC'15*, pp. 617–630 (2015).
- [4] Oukid, I., Lasperas, J., Nica, A., Willhalm, T. and Lehner, W.: FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory, *Proc. SIGMOD'16*, Association for Computing Machinery, pp. 371–386 (2016).
- [5] Kim, W.-H., Seo, J., Kim, J. and Nam, B.: ClfB-Tree: Cacheline Friendly Persistent B-Tree for NVRAM, *ACM Trans. Storage*, Vol. 14, No. 1, pp. 1–17 (2018).
- [6] Larus, J. and Kozyrakis, C.: Transactional Memory, *Commun. ACM*, Vol. 51, No. 7, pp. 80–88 (2008).
- [7] Intel: Intel®64 and IA-32 architectures software developer's manual volume 1: Basic Architecture, Intel Corporation (online), available from (<https://software.intel.com/en-us/articles/intel-sdm>) (accessed 2020-01-05).
- [8] Intel: Intel®64 and IA-32 architectures software developer's manual volume 2: Instruction Set Reference, A-Z, Intel Corporation (online), available from (<https://software.intel.com/en-us/articles/intel-sdm>) (accessed 2020-01-05).
- [9] Comer, D.: Ubiquitous B-Tree, *ACM Comput. Surv.*, Vol. 11, No. 2, pp. 121–137 (1979).
- [10] Jannink, J.: Implementing Deletion in B⁺-Trees, *SIGMOD Rec.*, Vol. 24, No. 1, pp. 33–38 (1995).
- [11] Chen, S., Gibbons, P. and Nath, S.: Rethinking Database Algorithms for Phase Change Memory, *Proc. CIDR'11*, pp. 21–31 (2011).
- [12] Jian, X. and Steven, S.: NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories, *Proc. FAST'16*, USENIX Association, pp. 323–338 (2016).
- [13] Venkataraman, S., Tolia, N., Ranganathan, P. and Campbell, R. H.: Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory, *Proc. of FAST'11*, USENIX Association, p. 5 (2011).
- [14] Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dullloor, S. R., Zhao, J. and Swanson, S.: Basic Performance Measurements of the Intel Optane DC Persistent Memory Module (2019).