同時マルチスレッディング環境下における永続メモリ向けスケジューラの検討

貴田 駿[1,2,a)]　広渕 崇宏[2,b)]　高野 了成[2,c)]　河野 健二[1,d)]

概要：バイト単位のアクセスが可能な不揮発性を持つ記憶デバイスである Persistent Memory が実用化の段階に入っている．Persistent Memory は DRAM と比較し，アクセス遅延が長いこと加え，データを永続化させるための付加的な処理が必要である．そのため，CPU のストール時間が長くなる傾向にあり，計算リソースが遊休状態となる時間が長い．通常の二次記憶デバイスとは異なり，Persistent Memory へのアクセスはオペレーティングシステムを介さないため，通常の方法でスケジューリングを行うことも容易ではない．本論文では，Persistent Memory へのアクセスによるストール中であっても，CPU の計算リソースを有効に活用するため，同時マルチスレッディング (SMT) を利用する手法を検討する．Persistent Memory に頻繁にアクセスを行うタスクと，CPU の計算リソースを多用するタスクとを同一物理コアに割り当てるようにする．これにより，CPU のストールにより利用されない計算リソースを自動的に他方のタスクに割り当てることができ，計算リソースが遊休状態になるのを避けることができる．この方式の有効性を確認するため，計算主体のタスクとして SPEC CPU 2017，Persistent Memory へのアクセスを行うタスクとして，Intel PMDK (Persistent Memory Development Kit) に付属の永続データ構造にアクセスするタスクを利用した実験を行った．その結果，提案方式により 6〜16 %程度のスループットの改善，および 8〜28 %程度の実行時間の改善が確認できた．

Shun Kida[1,2,a)]　Takahiro Hirofuchi[2,b)]　Ryousei Takano[2,c)]　Kenji Kono[1,d)]

1. はじめに

バイト単位でのアドレッシングが可能でありながら永続性を持つ永続メモリ（Persistent Memory）が実用化されるようになっている．たとえば，Intel 社の Optane Persistent Memory (PM) [1] はその一例である．永続メモリのアクセス遅延時間は DRAM のアクセス遅延時間に匹敵するといわれているものの，Intel PM を対象とした多くの性能評価実験では，実際のアクセス遅延時間は DRAM の 96 ナノ秒に比べ 391 ナノ秒程度となっており，その遅延時間を無視することはできない [5]．また，実際に製品化された永続メモリでは，メモリ内部でバッファリングやウェアレベリング等の処理を行うため，記憶デバイスそのものが持つ性能特性とは大きく異なる性能特性を示すことが知られている [8]．加えて，アクセス・パターンによる遅延時間の変動が DRAM のそれとは大きく異なるため，DRAM の遅延時間隠蔽のために用いられてきたさまざまな最適化手法をそのまま適用することは難しい．

本論文では，既存の CPU が提供する同時マルチスレッディング (simultaneous multithreading; SMT) の機能を活用することで，永続メモリのアクセス遅延を隠蔽する手法を検討する．同時マルチスレッディングとは，CPU の持つ物理コアを複数の論理コアとして見せる CPU の機能であり，Intel 社の CPU などではハイパースレッディングなどと呼ばれている．論理コア上で実行されるハードウェア・スレッドは，物理コアの持つ演算ユニット等を共有するため，同時に実行するスレッド間で演算ユニット等の衝突がなければ，高いスループットを提供できる．その反面，演算ユニットの衝突が頻発すれば，頻繁に演算がストールすることとなりスループットが低下する懸念がある．そのため，同一の物理コアで実行するスレッドを適切に選択し

1 慶應義塾大学
Department of Information and Computer Science, Keio University
2 国立研究開発法人産業技術総合研究所
National Institute of Advanced Industrial Science and Technology (AIST)
a) shunkida@sslab.ics.keio.ac.jp
b) t.hirofuchi@aist.go.jp
c) takano-ryousei@aist.go.jp
d) kono@sslab.ics.keio.ac.jp

ⓒ 2021 Information Processing Society of Japan

スケジューリングする必要があるものの，スレッドの必要とする演算ユニット等を事前に見積もることが難しく，適切なスケジューリング手法は知られていない．

本論文で検討する手法では，1) 永続メモリへのアクセスを頻繁に行うスレッドと，2) 永続メモリへのアクセスは行わない CPU インテンシブなスレッドを同一の物理コアで実行する．これにより，1) のスレッドが永続メモリへのアクセスによりストールしている間，2) のスレッドはすべての演算ユニットを利用することができる．その結果，全体としてより高いスループットを達成できる．論理コアで実行するスレッドの特性を考慮せずに，たとえば 1) のように永続メモリへのアクセスを頻繁に行うスレッド同士を同一物理コア上で実行してしまうと，永続メモリへのアクセスによるストール中は演算ユニットが活用されていない状態となる．逆に 2) の CPU インテンシブなスレッド同士を同一物理コア上で実行してしまうと，演算ユニットの衝突が起こり，高いスループットを得ることができない．これ以降，1) のように永続メモリへのアクセスを頻繁に行うスレッドを *IO-intensive task* と呼び，2) のように CPU による計算主体のスレッドを *CPU-intensive task* と呼ぶ．

実際に Intel 社の Optane Persistent Memory を用いて，次の 3 つの組み合わせを用いてスループットの計測を行なった．1) IO-intensive task と CPU-intensive task を同一の物理コアで実行する組み合わせ，2) IO-intensive task と IO-intensive task を同一の物理コアで実行する組み合わせ，3) CPU-intensive task と CPU-intensive task を同一の物理コアで実行する組み合わせである．IO-intensive task としては，Intel 社が提供している Optane Persistent Memory 用の永続データ構造（赤黒木）を利用し，また，CPU-intensive task としては SPEC CPU 2017 を用いている．詳細な実験環境については 4.1 節で述べる．この結果，SMT を有効にした上で IO-intensive task と CPU-intensive task を同一の物理コアで実行する組み合わせで実行することで両方のタスクともスループットが最も高くなることが確認できた．

本論文の結果は，同時マルチスレッディングの仕組みを用いることで，永続メモリを意識したスケジューリングが可能であることを示唆している．永続メモリに比べてアクセス遅延の短い DRAM では，メモリアクセスによるストール時間が相対的に短いため，同様の手法を適用しても十分に効果を得ることはできない．一方，SSD やハードディスクなどの二次記憶装置にアクセスする場合は，オペレーティングシステムによるスケジューリングが可能である．永続メモリへのアクセスにはオペレーティングシステムが介在しないため，オペレーティングシステムによる直接的なスケジューリングは行うことはできないものの，同時マルチスレッディングにより間接的にスケジューリングが可能であることを示している．

本論文は次のように構成されている．2 章では， Persistent Memory 特性を説明し，計算リソースの使用効率の観点における問題点を説明する．3 章では，本研究の目的とアプローチについて説明する．4 章では，2 つの実験をし，本研究のアプローチによってプログラムの実行性能が向上していることを示す．5 章では，関連研究を紹介する．6 章では，本論文のまとめを述べる．

2. 背景

近年実用化された，永続メモリ（Persistent Memory）の特徴について説明する．Persistent Memory は HDD・SSD のような不揮発性を持ち，HDD・SSD より高速で，DRAM よりやや遅い程度の高速なバイト単位のアクセスが可能な記憶デバイスである．また，大容量化が容易で DRAM より容量単価が安くできるというメリットもある．これらの特徴から，Persistent Memory は Key-Value Store やデータベース，科学計算などにおいて活用されることが期待される．

ただし， Persistent Memory のデータを永続化するには，データの一貫性を保ちながら FLUSH/FENCE を慎重に行う必要があり，DRAM と比較すると，遅延時間の差よりも大きく性能の制約を受けてしまう．このような遅延時間や，永続化のための処理中には CPU の計算リソースはあまり使われていない．DRAM にも当然このような時間は存在するが，Persistent Memory は上記の理由からより多くの時間 CPU の計算リソースを無駄にしていることになる．

CPU 計算リソースが使われていない時間を減らす単純なアプローチとして，演算を多く行うようなプロセスにコンテキストスイッチすることが考えられる．しかし，遅延がマイクロ秒未満である Persistent Memory へのアクセスの度にコンテキストスイッチしてしまうとそれ自体が性能のボトルネックとなってしまい逆効果だと考えられる．

3. 本研究の目的・提案

本研究の目的は，Persistent Memory の DRAM より長いアクセスの待ち時間に，CPU の整数演算器や浮動小数演算器の計算リソースを有効に活用することである．ただし，2 章で述べたようにアクセスの度にコンテキストスイッチを行うと，それが大きなオーバーヘッドを発生させてしまうため避ける必要がある．

この目的を達成するために, Simultaneous Multi Threading (SMT) を利用する．SMT を有効にすることによって Persistent Memory へのアクセスの待ち時間に，別の論理コアでコンテキストスイッチをすることなく使われていない計算リソースを利用するタスクを実行することができる．

SMT を利用することによってアクセスの待ちをしている裏で計算リソースを利用することができるが，より効率よく

ⓒ 2021 Information Processing Society of Japan

CPU の計算リソースを使うためには不十分である．なぜなら，同一の計算リソースを利用するようなタスクを単一の物理コアに割り当てるとリソースの競合が発生して使用効率が悪いためである．計算リソースを最も効率よく使用するためには，リソースの競合を避けて Persistent Memory へのアクセスを多く行うようなタスクと整数演算器や浮動小数演算器を多く利用するようなタスクを単一の物理コアに割り当てるのがよい．結果として，Persistent Memory の読み書きスループットと CPU-intensive なタスクの実行速度がともに向上すると考えられる．以降，メモリデバイスへのアクセスを多く行うタスクを IO-intensive task，CPU の演算器を多く使用するタスクを CPU-intensive task と呼ぶことにする．

SMT において計算リソースの競合を避けるアプローチで高速化を図っている先行研究は存在する．Persistent Memory のアクセス待ち時間は DRAM より長いため，よりリソースの競合が深刻な問題になってしまう．そのため， DRAM ベースのアプリケーションに適用させるよりもリソース競合を避けることの効果が大きくなる．現状，Persistent Memory ベースのアプリケーションに適用したものはない．

4. 評価

本研究のタスクのコア割り当てのアプローチの検証を行う．Persistent Memory へのアクセスと整数演算や浮動小数演算を同時に行う場面で，SMT を無効にした場合と有効にした場合を比較して，有効の場合の方がアクセス性能・計算性能がともに高くなることを検証する．また，タスクを様々な配置でコアに割り当てる．このとき，リソースの競合を避けるような割り当て方でアクセス性能・計算性能が最高になることを検証する．

4.1 実験概要・環境

この節では，検証を行うための実験の概要を説明する．

今回の実験では 2 つの物理コアを使用した．SMT を有効にすると 1 つの物理コアに対してそれぞれ 2 つの論理コアが存在している．IO-intensive task と CPU-intensive task を用意して，タスクのコアへの割り当て方を変えて実行し，IO-intensive task のスループットと CPU-intensive task の実行速度を測定して比較した．実験の主なコアへのタスクの割り当て方の設定を，図 1 に示す 3 つ用意した．

- cross: 単一の物理コアに対応する 2 つの論理コアに対してそれぞれ IO-intensive task と CPU-intensive task を割り当てる．本研究の提案する設定である．
- parallel: 2 つの物理コアに対して， IO-intensive task と CPU-intensive task を 2 つずつ割り当てる．
- nosmt: SMT を無効にして， 2 つの物理コアに対して， IO-intensive task と CPU-intensive task を 1 つずつ割り当てる．

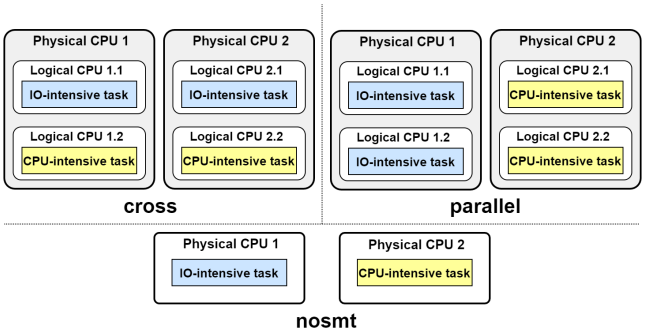

図 1: コアの割り当ての設定

表 1: 実験環境

CPU	
Type	Intel Xeon Gold 6230 CPU
Cores	40 cores across 2 sockets (use only 2 cores)
Frequency	2.10GHz
Caches	L1: 32KB Icache, 32KB Dcache L2: 1024KB L3: 28160KB
Memory	
PM Capacity	1.5TB (across 2 sockets)
DRAM Capacity	1.5TB (across 2 sockets)

これらの基本的な 3 つのタスクの割り当て方に加えて，SMT が有効の場合と無効の場合を同じスレッド数で比較するために， nosmt の設定でもそれぞれの物理コアに 2 つのタスク（スレッド）を割り当てる設定でも実行した．4 つのタスク割り当て方は cross と parallel と同様である．このとき，同一の物理コアに割り当てられた 2 つのタスクは OS によってスケジューリングされ，実行される．

以上のような設定で，独自の単純なタスクを用いた実験と， SPEC CPU 2017 [4] と PMDK [2] のサンプルを用いたより客観的で現実に近い実験の 2 つを行った．SPEC CPU を使った理由は現実に存在するタスク（コンパイル・圧縮・科学計算など）を用いているからである．また，PMDK を使った理由は Intel が提供している Persistent Memory を操作するライブラリであり，現状の Persistent Memory ベースのアプリケーションの多くでこれが使われているためである．

実験環境を表 1 に示す．今回の実験では物理コアを 2 つしか使わなく，同じソケット内のものを使うものとする．CPU の省電力設定は全て無効にして，常に最大周波数で動作するように設定した．また，実験結果に影響を与えないように，ハードウェア・プリフェッチャーも全て無効にした．

ⓒ 2021 Information Processing Society of Japan

4.2 単純なタスクによる検証

この節では，独自に用意した単純なタスクによる検証実験について述べる．

4.2.1 目的

独自に用意した作為的なタスクを使って，タスクのコアへの割り当て方の設定の比較を行う．本研究の提案である cross の設定が，IO-intensive task のスループット，CPU-intensive task の実行速度が他の設定と比べて良くなっているかを確認する．また， DRAM にアクセスする IO-intensive task と Persistent Memory にアクセスする IO-intensive task を考えられるが，本研究のアプローチは Persistent Memory に対して行う方がより効果があることを確認する．これは 2, 3 章で述べたように， Persistent Memory のアクセス待ち時間が DRAM より遅いことに起因する．

4.2.2 手法

独自に用意した IO-intensive task と CPU-intensive task と，それぞれのスループット・実行速度を測定する手法について説明する．

IO-intensive task としては, ひたすらメモリデバイスへアクセスを行い続けるものを用いた．操作は全て書き込みで，ランダムにアクセスを行った．アクセスの粒度に関しては，Persistent Memory の書き込みの粒度である 256B を用いた．アクセス対象のメモリデバイスは DRAM と Persistent Memory とし，この 2 つを比較する．DRAM へのアクセスは memmove を用いて書き込みを行い, Persistent Memory へのアクセスは PMDK の pmem_memmove_persist を用いて書き込みと FLUSH/FENCE まで行うことにした．

CPU-intensive task としては，プログラム全体のメモリアクセスが占めるの割合が異なるタスクを 3 つ用意した．というのも，メモリアクセスが占める割合が大きければ，それだけ CPU-intensive task としての傾向が弱くなってしまう．結果として IO-intensive task とのリソース競合が起こりやすくなり，計算リソースを十分に活かしきれないことになると考えられる．この傾向を確認するために以下の 3 つのタスクで測定を行い，比較した．それぞれのタスクの量を変化させて，その実行時間から実行速度を推測した．

(a) IDIV 命令を繰り返し実行するタスク．メモリアクセスをしないようにアセンブリで記述した．
(b) 行列積を計算するタスク．行列が DRAM 上に存在するためメモリアクセスが行われる．
(c) 行列和を計算するタスク．プログラム全体に占めるメモリアクセスの割合が (b) よりも大きい．

プログラム全体の流れを説明する．まず，IO-intensive task と CPU-intensive task を実行するスレッドを生成して，指定のコアに固定する．この際， IO-intensive task と CPU-intensive task は 1 つずつ組にしておく．それぞれ実行の前準備（行列の生成など）を行った後，全てのスレッドを同期させて同時にタスクを開始する．CPU-intensive task の計算が終わったらスレッド間で共有するフラグ変数を建てて，それを監視する，組になっている IO-intensive task も終了する．全てのタスクが終了した後, IO-intensive task のスループットは足し合わせ，CPU-intensive task の実行時間は平均を取って測定結果として出力する．

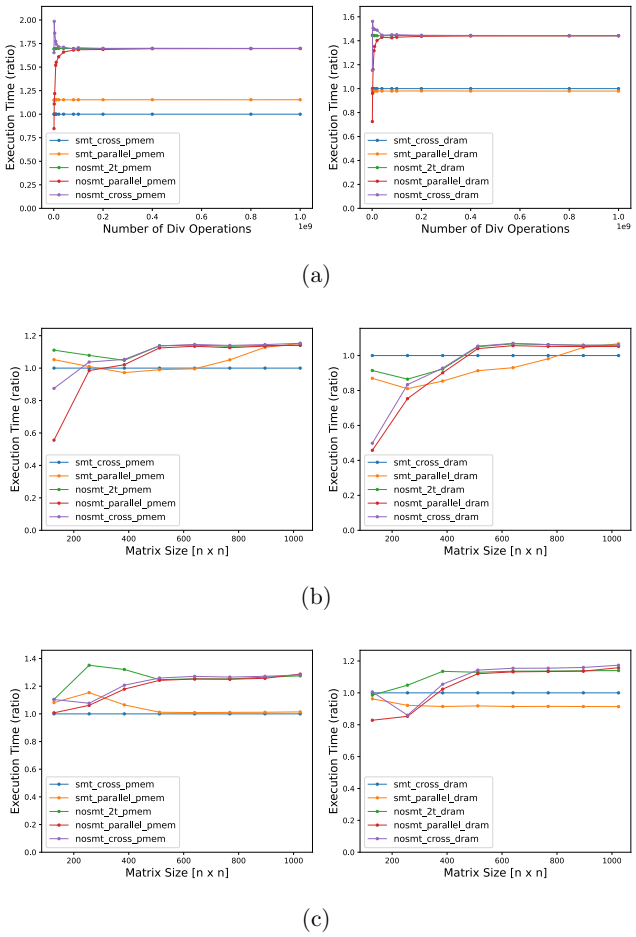

図 2: 各 CPU-intensive task の実行時間の比の値の比較

4.2.3 結果

まずは，(a) (c) それぞれの CPU-intensive task について，IO-intensive task が PMEM・DRAM 両方の場合の実行時間のグラフを図 2 に示す．実行時間は SMT を有効・コアの割り当てを cross にした smt-cross の時の値を 1 として比の値を用いている．実行時間が短いほど，CPU-intensive task の処理速度が速いことを意味する．横軸はタスク (a) では IDIV 命令の数，(b),(c) では正方行列の一辺の大きさである．

図 2 (a) の IO-intensive task が Persistent Memory への書き込みを行う場合（左側）を見ると，smt-cross の設定で最も実行が速くなっている．smt-cross は smt-parallel より 15 % 程度，nosmt の 3 つの設定よりも 70 % 程度高速となった．DRAM への書き込みを行う場合（右側）を見

ⓒ 2021 Information Processing Society of Japan

ると，smt-cross は smt-parallel より 2 % 程度遅く，nosmt よりも 44 % 程度高速となっている．これは， DRAM は Persistent Memory よりも遅延時間が短く，永続化のための処理もないため，裏で動いている CPU-intensive task と CPU の計算リソースの競合を起こしているためだと考えられる．

同様に 2 (b) のグラフを見ると，Persistent Memory と DRAM の場合ともに smt-cross が最速となっている．行列のサイズが小さい時にグラフが大きく乱れているが，これは実行時間が短く同期時のずれなどの影響が大きくなっているためだと考えられる．そのため，大きな行列の時の結果を比較すると， smt-cross は他の設定より，Persistent Memory への書き込みを裏で行っている場合は 15 % 程度，DRAM の場合は 7 % 程度高速になった．

最後に 2 (c) のグラフを見ると，裏で Persistent Memory への書き込みを行っている場合は smt-cross と smt-parallel のグラフが重なっている．nosmt の場合と比べると， 27 % 程度高速になっている．裏で DRAM への書き込みを行っている場合には smt-cross と smt-parallel の処理時間が逆転している．結果， smt-cross は smt-parallel より 8 % 程度遅く， nosmt よりも 14 % 程度高速になっている．

同様に，各 CPU-intensive task と組み合わせた場合の IO-intensive task のスループットのグラフを図 3 に示す．

図 3 の全てのグラフにおいて smt-cross のスループットが最も高くなっている．裏で Persistent Memory への書き込みを行っている場合（左側）では，smt-cross のスループットは smt-parallel より 7-8 % 程度高く，nosmt の 2 倍程度になっている．このことから，SMT により Persistent Memory の遅延時間や FLUSH/FENCE などの永続化のための処理を非常に上手く隠蔽できていると考えられる．

裏で DRAM への書き込みを行っている場合（右側）では，smt-cross のスループットは smt-parallel より 25 % 前後高く，nosmt より 50 % 前後高くなっている．Persistent Memory の場合と比べて，SMT 有効化による改善が半分程度になっている一方で，タスクの配置の最適化によるスループット上昇が大きくなっていることがわかる．これは，DRAM の場合は工夫もしなくても，遅延時間は大したスループット低下に起こさず，別の物理コアに IO-intensive task を置いたことによりメモリコントローラーによるスループット低下を改善したためだと考えられる．

以上の結果から，次の 2 つのことが確認できた．

- 基本的に，Persistnet Memory の方が DRAM ベースのプログラムよりも SMT 有効化・タスクの配置の最適化の効果が大きくなる．すなわち，Persistent Memory にこそ本研究のアプローチを適用することに意味がある．
- より CPU-intensive なタスクと IO-intensive task を組み合わせた方が，タスクの配置の最適化による効果が大きくなる．

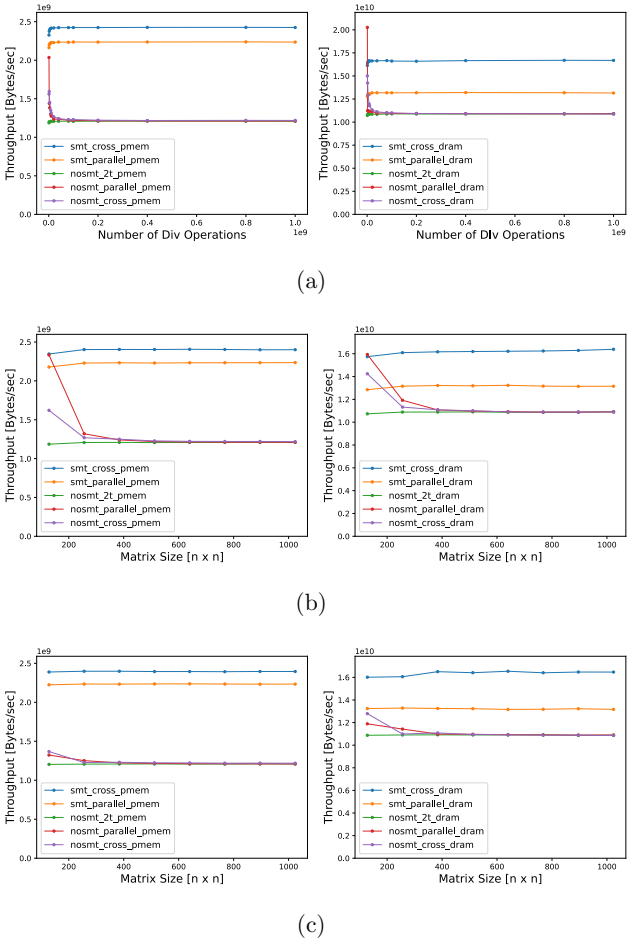

図 3: 各 CPU-intensive task と組み合わせた場合の IO-intensive task のスループットの比の値の比較

4.3 SPEC CPU + PMDK サンプルによる検証

この節では，SPEC CPU 2017 と PMDK のサンプルをタスクとして用いた実験について述べる．

4.3.1 目的

4.2 のような作為的なベンチマークではなく，客観的なタスクを用いることで本研究のアプローチの効果が現実的であることを示す．SPEC CPU は，様々な整数系・浮動小数点数系のプログラムを含み，CPU やメモリの性能を評価するベンチマークである．PMDK は Intel が提供する Persistent Memory 向けのライブラリで，多くの不揮発性メモリベースのアプリケーションで用いられている．したがって，この 2 つを用いたタスクを用いた実験は客観的であると考えられる．

4.3.2 手法

SPEC CPU 2017 と PMDK のサンプルを用いて，それぞれの実行速度やスループットを測定する手法について説明する．

CPU-intensive task としては， SPEC CPU 2017 のそ

ⓒ 2021 Information Processing Society of Japan

れぞれのベンチマークを利用した．SPEC CPU 2017 には，整数系が 10，浮動小数点数系が 14 のベンチマークが含まれる．また，それぞれには rate と speed という測定方法が存在する．rate が一定時間にどれだけのワークロードを処理出来るかを測定するのに対して，speed はワークロードを完了するのにかかる時間を測定する．今回は実行速度を求めるために rate を使用した．ただし，rate の設定で実行できないベンチマークは省いた．それぞれのベンチマークは事前にビルドしておき，測定時には余計な処理を含まないように実行した．

IO-intensive task としては，PMDK のサンプルを用いた．具体的には PMDK の中で，オブジェクト単位でデータ管理し，トランザクションやロックをサポートする libpmemobj というライブラリのサンプルを利用した．複数のサンプルの中で，今回は赤黒木に対して挿入削除をトランザクショナルに行えるようなサンプルのライブラリ [3] を利用した．これは， Persistent Memory の想定される用途の Key-Value Store を意識したためである．サンプルそのままでは指定個数のノード挿入をして終了してしまうため，CPU-intensive task 実行中はずっとアクセスを操作を続けるように変更を行った．

プログラム全体の流れの概要を図 4 に示す．まず，IO-intensive task から実行を開始し，赤黒木をベースとしたマップに対して一定個数（今回は 1000）のノードを挿入する．このステップを 4 では warm up として表現した．IO-intensive task は warm up を終えると，マップに対してランダムに削除と挿入を繰り返す．これは，測定を行っている間にマップ内のノード数を一定に保ち，安定的に Persistent Memory へのアクセスをするためである．測定中にノードの数が変化すると，挿入・削除の操作にかかる時間も変化してしまい公平な測定が難しい．IO-intensive task は削除・挿入の間，CPU-intensive task 側からユーザ定義シグナル SIGUSR1 を受け取ると測定を開始したと認識する．その後， CPU-intensive task（すなわち SPEC CPU ベンチマーク）の修了時に別のユーザ定義シグナル SIGUSR2 が渡される．SIGUSR1 を受け取ってから SIGUSR2 を受け取るまでの処理した挿入・削除の操作数，時間を測定する．そこから，単位時間あたりに処理した挿入・削除の操作数を計算することができ，これを IO-intensive task のスループットと考えた．CPU-intensive task の実行速度は，SPEC CPU がテスト対象システムでの実行を基準としたスコアを出力するため，これを利用する．

4.3.3 結果

SMT を無効化した場合の設定として，無理やり 1 つのコアに複数のタスク（スレッド）を割り当てて OS のスケジューラに実行を任せた上で cross と parallel と同じ割り当てでも測定を行った．これらの設定では，OS のスケジューラが IO-intensive task と CPU-intensive task をバランスよく実行することが出来ていなかったため，図 5,6 から省いた．nosmt-cross の場合，殆どの CPU 時間を CPU-intensive task に費やしてしまった．逆に， nosmt-parallel の場合は，CPU-intensive task より多くの CPU 時間を IO-intensive に費やしていた．

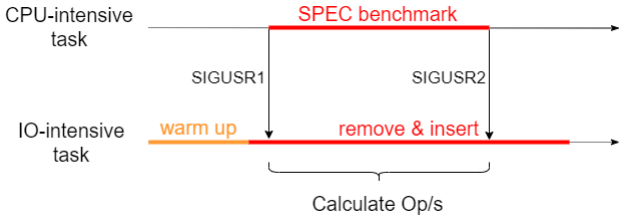

図 4: SPEC CPU と PMDK サンプルを用いた測定の流れ

図 5 に SPEC CPU 2017 の 23 個のベンチマークそれぞれに伴って実行した，PMDK サンプルをベースにした IO-intensive task のスループットを示す．全てのベンチマークにおいて，程度の差はあるが smt-cross が最も高いスループットを示した．また，同じく全てのベンチマークにおいて， nosmt よりも smt-parallel の方が高いスループットを示した．したがって，Persistent Memory へのアクセス・スループットは SMT を有効にすることで大きく改善し，CPU-intensive なタスクに伴って対応する論理コアに割り当てることによってさらにスループットが改善することがわかった．

ベンチマーク間で smt-cross と smt-parallel の間に多少の差が見られる．4.2 節の結果とあわせて考えると，2 つの設定の差が小さいものは，ベンチマークに含まれるメモリアクセスの量が多いと推測ができる．

図 6 に SPEC CPU 2017 の各ベンチマーク側の実行速度を示す．こちらも，概ね全てのベンチマークにおいて実行速度が高い順に，smt-cross, smt-parallel, nosmt-2t となっている．一部のベンチマークで smt-parallel と nosmt-2t の差が殆どない，または反転する現象が起きている．また，smt-cross の他の設定に対する実行速度の改善率も，スループット以上にばらつきが多い．これは，SPEC CPU 側は Persistent Memory へのアクセスはしないが，IO-intensive task 側は CPU の計算リソースを使用するため，こちらのリソースの方が競合しやすいのだと考えられる．Persistent Memory へのアクセスを多く行うタスクと CPU-intensive なタスクを利用するとき，SMT を有効にした上で，コアへのタスクの割り当て方を工夫することで CPU-intensive タスクの実行速度の改善が見込めることがわかった．

5. 関連研究

同時マルチスレッディングにおいて，同一物理コア上で実行するワークロードの特性により，レイテンシおよびスループットが大きな影響を受けることはよく知られている．そのため，同一物理コア上に割り当てる（ソフトウェア上

ⓒ 2021 Information Processing Society of Japan

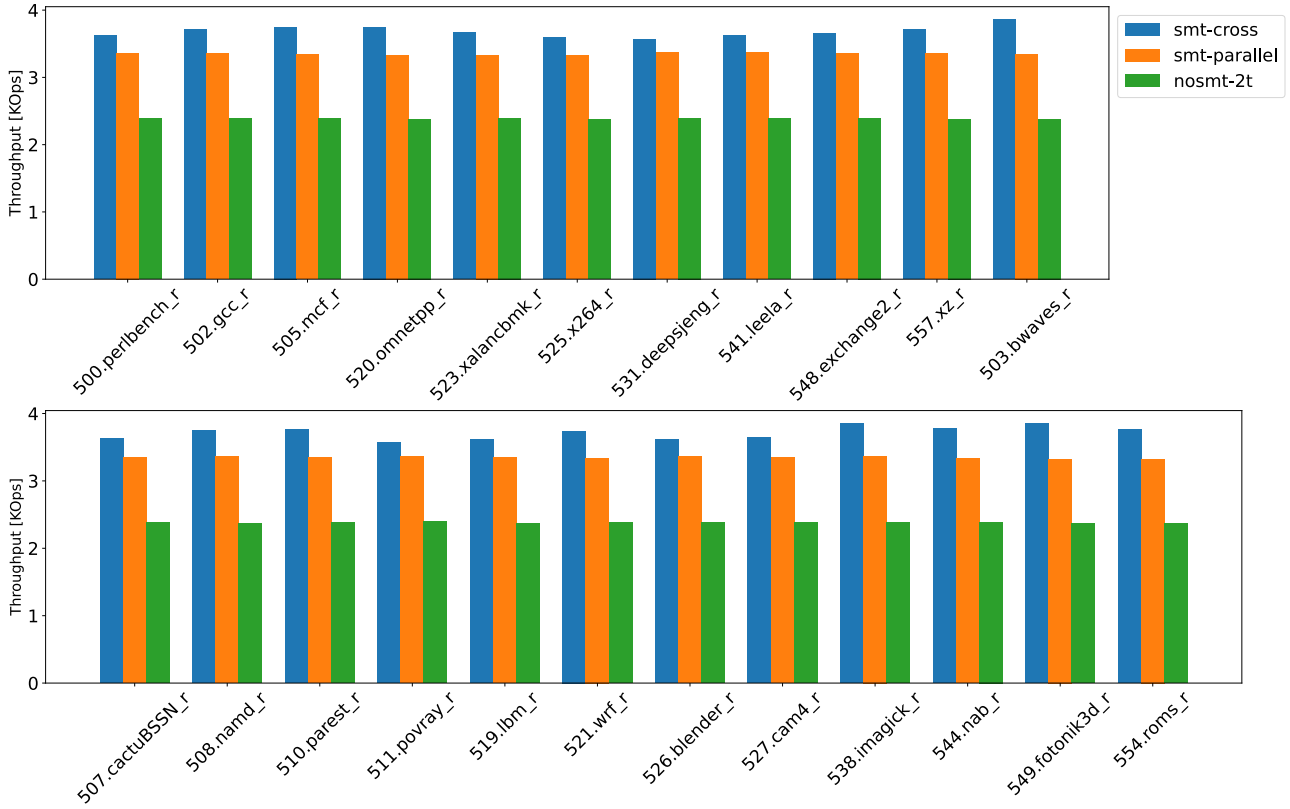

図 5: 各ベンチマークに伴って実行した IO-intensive task のスループット

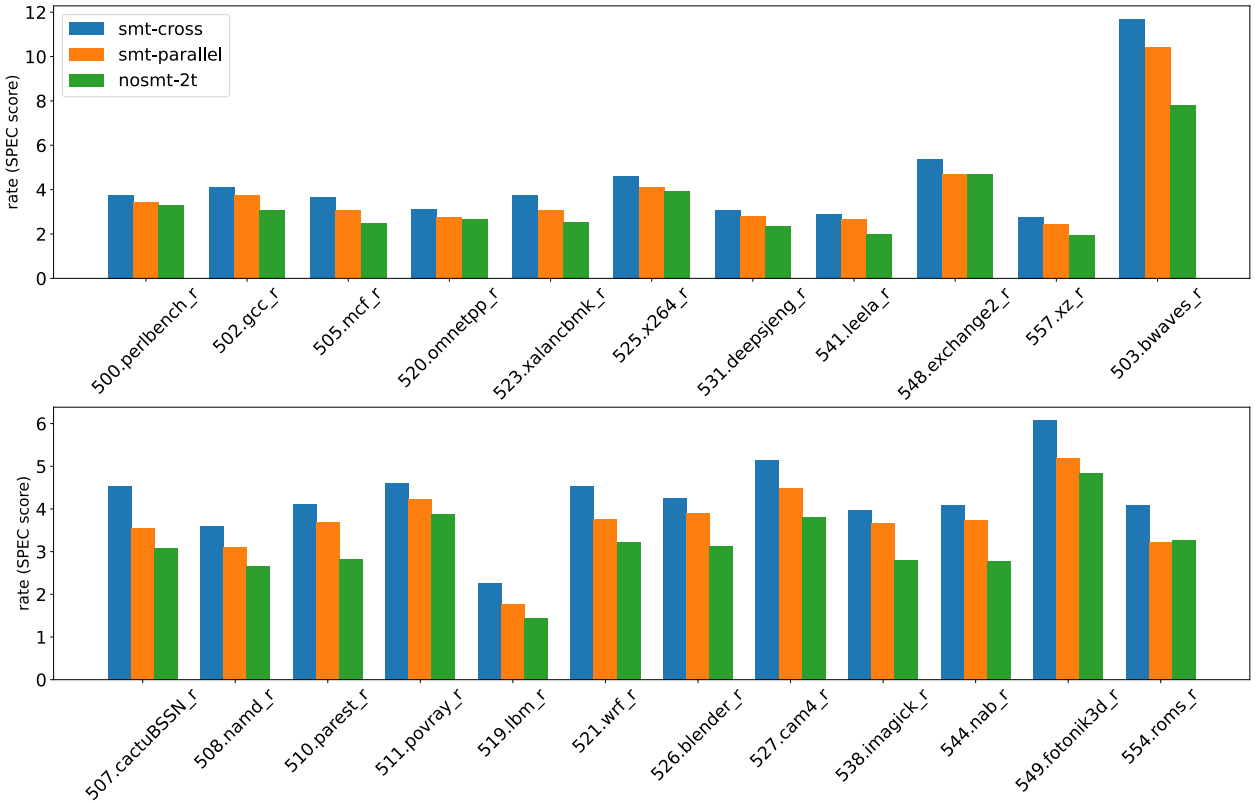

図 6: SPEC CPU 2017 の実行速度

ⓒ 2021 Information Processing Society of Japan

の）スレッドの組み合わせを工夫し，レイテンシやスループットの改善を行う手法は古くから試みられており，一般に共生スケジューリング (symbiotic scheduling) と呼ばれている [7]．共生スケジューリングの基本的な考え方は，パフォーマンス・カウンタなどのモニタリング結果から，ハードウェア・スレッド間での演算ユニット等の衝突を予測あるいは検知し，衝突を回避するように（ソフトウェアでの）スレッドとハードウェア・スレッドの割り当てを動的に変更するというものである．

文献 [7] で提案されている SOS というスケジューラでは，バッチジョブとレスポンス重視のジョブが混在した状況を想定し，レスポンス重視のジョブを阻害しないようにスケジュールするものである．Elfen スケジューリング [9] でも同様の状況を想定しており，この方式ではバッチジョブのコードを改変し，レスポンス重視のジョブの実行が始まったらハードウェア・スレッドを譲るようになっている．こうした従来の研究では永続メモリとの相性については考察されていない．本論文では，永続メモリと共生スケジューリングの考え方の相性が良いことを示したものである．

文献 [6] で提案されている vSMT-IO では，同時マルチスレッディング有効のプロセッサにおいてクラウドの典型的なワークロードを意識し，集中的な IO 操作と重い計算を行うときの IO-intensive task のスループットの向上を目的としている．同時マルチスレッディング有効のプロセッサでは，IO スループットを向上させるための既存の技術では非効率である．これは，ワークロードが IO 待ち時間に過剰なコンテキストスイッチやスピンを行うためである．vSMT-IO では，ハードウェアスレッドに IO ワークロードのコンテキストを保持することによってコンテキストスイッチやスピンのオーバーヘッドを取り除き，スループットの向上を図っている．この研究では，IO ワークロードが永続メモリへのアクセスである時の最適化については考察されていない．本論文では，コンテキストスイッチを避けた上で，集中的な永続メモリアクセスと重い計算を同時に行うときのスループット向上を目的としている．

6. まとめ

近年，不揮発節を持つバイト単位でのアクセスが可能な Persistent Memory が実用化されてきている．DRAM よりも長い遅延時間・永続化のための処理のために，CPU の計算リソースが無駄になっている時間が長い．単純にコンテキストスイッチをしてしまうと，むしろそれ自体が性能上のボトルネックとなってしまう．本論文では，Persistent Memory へアクセス中の CPU の計算リソースを有効活用するために，同時マルチスレッディング（SMT）を有効にした上で，Persistent Memory へのアクセスを多く行うタスクと CPU の計算リソースを多く使うを同じ物理コアに割り当てることを提案した．こうすることで単一物理コア内のリソースの競合が起こりにくく，両方のタスクの性能が向上した．独自のタスクを用いた実験と，SPEC CPU 2017 と PMDK のサンプルをベースにしたタスクを用いた実験の 2 つを行った．その結果，SMT を無効にした場合やコアへのタスクの割り当て方を工夫しなかった場合に比べ，本論文の提案手法が高い性能を示すことが確認できた．

謝辞　本研究の一部は科研費 19H01108 の助成を受けた．

参考文献

[1] : Intel Optane DC Persistent Memory, https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/optane-dc-persistent-memory.html.
[2] : Persistent Memory Development Kit, https://pmem.io/pmdk/.
[3] : rbtree.c, https://github.com/pmem/pmdk/blob/master/src/examples/libpmemobj/tree_map/rbtree_map.c.
[4] : SPEC CPU 2017, https://www.spec.org/cpu2017/.
[5] Hirofuchi, T. and Takano, R.: A prompt report on the performance of Intel optane DC persistent memory module, *IEICE Trans. Inf. Syst.*, Vol. E103.D, No. 5, pp. 1168–1172 (2020).
[6] Jia, W., Shan, J., Li, T. O., Shang, X., Cui, H. and Ding, X.: vSMT-IO: Improving I/O Performance and Efficiency on {SMT} Processors in Virtualized Clouds, *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pp. 449–463 (2020).
[7] Snavely, A. and Tullsen, D. M.: Symbiotic jobscheduling for a simultaneous multithreaded processor, *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS IX, New York, NY, USA, Association for Computing Machinery, pp. 234–244 (2000).
[8] Wang, Z., Liu, X., Yang, J., Michailidis, T., Swanson, S. and Zhao, J.: Characterizing and Modeling Non-Volatile Memory Systems (2020).
[9] Yang, X., Blackburn, S. M. and McKinley, K. S.: Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading, *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pp. 309–322 (2016).

ⓒ 2021 Information Processing Society of Japan