Vol.2018-OS-144 No.11
2018/7/31

POWER システムへのユーザレベル NVMe ドライバの移植と性能評価

吉村　剛[1,a)]　千葉　立寛[1,b)]　堀井　洋[1,c)]

概要：POWER システムはハードウェアの性能を最大限引き出し，エンタープライズデータベースや人工知能などの高い計算能力を要するワークロードを最適化する．しかし，近年のオープンソースソフトウェアは x86 CPU で開発・最適化される傾向があり，特に Non-volatile memory Express (NVMe) のような高性能ハードウェアのためのユーザレベルデバイスドライバにおいてアーキテクチャ固有の最適化が増えてきている．本研究はユーザレベルの NVMe ドライバのひとつである， Storage performance development kit (SPDK) を POWER システムへ移植し，必要となった最適化および Yahoo! Cloud Serving Benchmarks (YCSB) での実験結果を示す．移植ではページサイズ， IOMMU，メモリモデル, SIMD 命令の 4 種類のアーキテクチャレベルの違いを考慮する必要があり，特にメモリモデルは POWER での SPDK の性能にとって重要な違いであった．実験では MongoDB にバインドした RocksDB へ SPDK を適用し，最適化の結果 POWER でのデータロードのスループットが 76% 向上した．また SPDK により，RocksDB はデータロードのスループットが最大 10% 改善していた．

1. はじめに

POWER システムは x86 とは異なる独自の CPU アーキテクチャを採用しており，メモリや PCI Express などの，各種計算資源をより高い性能を引き出すためのプラットフォームとなっている．2018 年以降に提供が開始された POWER9 は，PCI Express Gen4 のサポートにより，エンタープライズデータベースや人工知能など，高い計算能力を要するワークロードにおいて，より広いバス帯域幅を持つ高性能 I/O デバイスの活用を可能にする．

しかし，データベースや人工知能を含め，多くのソフトウェアが x86 CPU 上で開発されており，暗黙のうちに x86 CPU に最適化された実装になっている．例えば仮想化機能に関する移植においても， x86 では十分な性能を引き出せていたエミュレーションベースの nested virtualization が ARM では性能が低下することが指摘されている [1]．このように，性能劣化をいかに防ぐかがソフトウェアの移植性に関する主要なトピックとなっている．

性能劣化は特に高性能ハードウェアの性能を引き出すことを目的としたソフトウェアの移植で重大な問題となる．高性能 I/O デバイスにおいてユーザレベルのデバイスドライバを利用することで性能を最大限引き出せるとされ [2]，その場合ユーザレベルのアーキテクチャ依存コードの移植が必要となる．例えば Storage Performance Development Kit (SPDK) [3] は IOMMU を利用することで， セキュアな Non-volatile memory Express (NVMe) のユーザレベルデバイスドライバ実装を可能にしている．しかし，SPDK は当初は x86 CPU およびインテル製の NVMe の利用を主に想定したフレームワークであり，POWER システムを含め多くのプラットフォームでの利用が十分サポートされていない．

本研究はユーザレベル NVMe ドライバのひとつである SPDK を POWER システムへ移植し，必要となった最適化および Yahoo! Cloud Serving Benchmarks (YCSB) [4] での実験結果を示す．実験では SPDK を MongoDB に組み込み，POWER8 および Haswell プラットフォームでの測定結果を示す．移植にはアーキテクチャの違いが原因となる修正として大きく分けて， 1) メモリページサイズの大きさの違いによるもの，2) IOMMU の違いによるもの，3) メモリモデルの違いによるもの，4) SIMD 命令の 4 つが必要となった．

特に，3) と 4) に関しては SPDK を動作させる分には必須ではないものの，x86 システムでの性能に近づけるための最適化として重要な修正となっている．実験でも 3) と 4) の最適化により POWER において， SPDK を用い

[1] IBM 東京基礎研究所
[a)] tyos@jp.ibm.com
[b)] chiba@jp.ibm.com
[c)] horii@jp.ibm.com

た MongoDB のデータロードのスループットが 76% 向上した．Haswell で動作させた場合に比べ，最適化前は 58% のスループット低下が見られたものの，最適化により 26% のスループット低下まで抑えることができた．最適化の結果，データロード以外にも Haswell に対し，POWER では YCSB の Workload A (read 50%, update 50%) は 26% の低下，Workload B (read 95%, update 5%) では 3%, Workload C (read 100%) では 6% のスループットの向上が見られた．

最適化の評価に加え，本研究は移植した SPDK の性能と他のデータベースワークロードとして，RocksDB をバインドした MongoDB, WiredTiger をバインドした MongoDB, Redis の計 4 種類の性能と比較する．実験結果から，RocksDB は SPDK を有効にすることで，データロードのスループットが最大 10% 改善し，その他の Workload B でも 4% のスループット改善が見られた．SPDK と他のデータベースワークロードの比較結果では，Workload C において，WiredTiger と比較して SPDK は最大 11% 高いスループットを示した．これら実験結果から，今後の最適化によりユーザレベル NVMe ドライバを POWER 環境で利用するアドバンテージを大きくしていくことができると期待される．一方で，WiredTiger は SPDK や RocksDB と比較して Workload A において 22% 高いスループットを示すなど，データ書き込みに関しては RocksDB などデータベースを含めた最適化をしていく必要があることも予想される．

本論文ははじめに 2 節で SPDK の内部実装についての概要を説明する．次に，実際に必要となった移植作業の詳細を 3 節で述べ，4 節では MongoDB に SPDK を組み込む実装の詳細を述べる．5 節では実験結果を述べる．6 節で関連研究をあげ，最後に 7 節でまとめと今後の課題について言及する．

2. SPDK

SPDK は NVMe フラッシュストレージを用いるワークロードにおいて，極めて低いレイテンシでかつ高いスループットでの I/O 処理をするために開発されたフレームワークである．NVMe は PCI Express を利用することから，元々はネットワークデバイスの高性能化を生かすための既存ライブラリである， Data Plane Development Kit (DPDK) [5] の PCI デバイスの初期化や低レベル操作の部分を SPDK は再利用している．特にユーザレベルのデバイスドライバをセキュアに作るために，IOMMU を操作する部分を再利用している．また，RocksDB のようなキーバリューストアへの SPDK の適用を簡単にするため，blobfs と呼ばれる既存のファイルシステムと類似したインターフェイスをユーザライブラリ内に持つ [6].

図 1 は SPDK の内部構造の概要を示している．データベースのクライアントから受け取った get や put，もしくは SQL クエリを受け取ったデータベースのサーバスレッドは，SPDK blobfs のインターフェイスへクライアントリクエストを変換して渡す．Blobfs は受け取ったリクエストをイベントとして内部にもつリングバッファへ一時保存する．リングバッファは DPDK のこれまでのメインターゲットであったネットワーク処理でも頻繁に使われるものであるため，SPDK は DPDK が既に提供している実装を再利用している．リングバッファへの書き出しはマルチスレッドで行うようになっており，ロックフリーで操作する実装にすることで高い性能を実現できている．Blobfs は SPDK のコアライブラリのもつスレッド機能を使って Reactor スレッドを生成しており，そのスレッドがイベントバッファへポーリングすることで，イベントをできるだけ低いレイテンシで処理するようにしている．Reactor スレッドは DMA 領域へ実際に必要となるデータを書き出しや読み出しを行なっており，PCI コンフィグ領域を利用して NVMe と直接やりとりする．PCI コンフィグ領域は memory mapped IO (MMIO) で操作でき，また DMA 領域も IOMMU とのマッピングを保持するため，ユーザ空間でもセキュリティを損なわずに DMA の制御をすることができる．IOMMU の初期化部分は DPDK が OS とのやりとりをカプセル化した API を持つため，それらを再利用している．

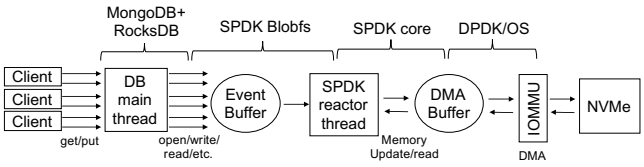

図 1 SPDK を利用した MongoDB, RocksDB の処理の流れ

このように，クライアントからのリクエストを受け取る部分以降，全ての処理は単一ユーザプロセス内で完結し，I/O 操作の大部分を占める DMA は全てユーザプロセスと NVMe デバイスの間で完結する．OS の役割は，初期化部分のアクセス制御とメモリ確保，実行時のスレッドスケジューリングが大部分となる．その結果，ユーザとカーネルのコンテキストスイッチはほとんどが排除され，極めて低いレイテンシでの NVMe デバイスとのデータのやりとりが可能となる．

3. SPDK の POWER8 への移植

SPDK は様々なアーキテクチャ固有のコードを含んでおり，POWER8 に移植するためには一部コードの変更が必須となる．本節では特に，両者のアーキテクチャの違いが SPDK およびそのバックエンドとして利用されている DPDK へもたらした問題について述べる．DPDK は公式には POWER へのサポートをしているとされているもの

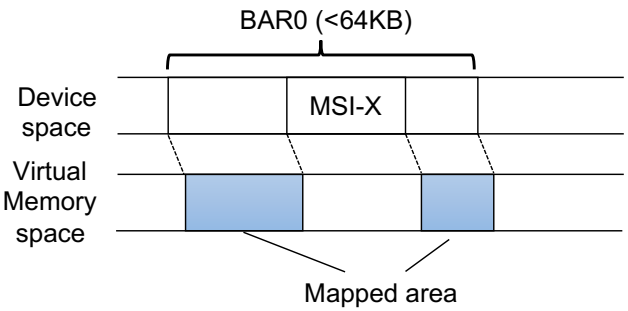

図 2　NVMe の BAR0 マッピング方法

の，元々はネットワークデバイスのためのライブラリであったためか，ストレージデバイスを利用した場合に問題が多く起きている．

3.1　ページサイズ

POWER は 64 KB メモリページをサポートしている一方で，x86 プラットフォームではラージページを除き全てのメモリページは 4KB として扱われる．その結果，多くのデバイスの内部で扱うページサイズも 4KB がひとつの単位として想定されることが多い．

ページサイズの違いによる最も大きな問題は，PCI デバイス制御のための BAR0 領域が割り込み制御のための MSI-X 領域の近傍に配置された場合，BAR0 マッピングが不可能になってしまうことである．MSI-X 領域は VFIO はマッピングすることはできない例外の領域として扱われており，BAR0 領域を読み書きするためには MSI-X 領域を除いた領域のみマッピングするワークアラウンドが必要となる (図 2)．しかし，64 KB ページでは BAR0 全体が 64KB 以下の大きさの場合，それが不可能となる．

その結果，現状の SPDK 移植はページサイズの対応のために Linux カーネルの変更を前提としている．そこでは MSI-X 領域のマッピングを許可しているものの，セキュリティ上の問題はまだ不透明であり，Linux の最新版ではまだマージされていない．今後 POWER での SPDK の本格的なサポートのためには異なるアプローチが必要になると考えているものの，本研究はこの問題を今後の課題として解決していない．

また，本研究で用いる NVMe のモデルは 4KB ページのみサポートしており，そのために SPDK でも問題を引き起こしていた．SPDK は他のシステムソフトウェア同様に，ページサイズ単位のメモリ確保が多数あるものの，ここではシステムのページサイズではなく 4KB のページサイズを利用し，デバイスと合わせる必要がある．

3.2　IOMMU

POWER の IOMMU は一部機能およびインターフェイスが x86 CPU と異なっており DPDK の変更が必要となる．x86 CPU の IOMMU は IOMMU Type 1 と呼ばれるのに対して，POWER による IOMMU は sPAPR と呼ばれている．Linux ではプラットフォーム非依存の Virtual Function I/O (VFIO) [7] がユーザレベルデバイスドライバにインターフェイスとして公開されており，いずれのアーキテクチャでもファイルディスクリプタに対する ioctl で IOMMU を操作することができる．しかし，ioctl の制御手順が異なっている．

DPDK は初期化の段階で，あらかじめヒュージページで確保された連続したメモリ領域を確保し，IOVA へマッピングし，デバイスとの DMA を準備する．アドレス計算の簡易化のため，DPDK は IO 仮想アドレス (IOVA) と物理メモリアドレスを同一のアドレスとなるようにし，仮想メモリアドレスへストレートマッピングしている．例えば物理メモリアドレス 0x100 のデータは IOVA 0x100 へ転送されるようにマッピングしている．全てのヒュージページ領域を DMA するわけではないものの，実装の簡易化のためにまとめて DMA 可能な領域としてピンしている．メモリピンは OS によるスワップアウトを防ぐために不可欠となる．

sPAPR は Type 1 IOMMU と異なり，マッピングの登録とピンを別々に行なっており，そのときに IOVA として利用可能なアドレス領域を事前に指定する必要がある．実際に移植したコードでは，0 番地アドレスからヒュージページとして確保されたページの物理ページで最大のアドレスを IOVA の領域としている．Type 1 IOMMU では複数の領域を別々にピンするのに対して，sPAPR では 1 つの巨大な領域をピンしていることになる．この理由として，単純にデバイス側でただひとつの領域のみサポートしているためである．DMA 領域としての登録は Type 1 IOMMU と同様に行うことができる．

3.3　メモリモデル

POWER の最大の特徴のひとつは “弱い” メモリモデルを採用していることである [8]．x86 CPU は，ある CPU によるある変数の読み書きの結果は，メモリ同期命令はなくとも他の CPU から認識することができる．このような “強い” メモリモデルは全ての CPU で採用されているわけではなく，ARM や POWER はユーザが明示的にメモリの同期命令を発行する必要がある．つまり，開発者が適切にメモリ同期を指定しなければバグや性能の劣化の原因となる．メモリモデルの違いは，特に高性能化を目指すような SPDK や DPDK のようなソフトウェアで重要となる．

3.3.1　リングバッファ

デバイスへの高速な転送はロックフリーのリングバッファを用いることが典型的な実装になる．しかし，ロックフリーの場合，x86 CPU とは異なり，メモリモデルの違いを意識した実装をしなくてはならない．なぜなら，ロッ

```
enqueue(prod, input) {
  do {
    oldH = prod.head
    lwsync
    newH = oldH + 1
    succ = cmp_and_set(prod.head, oldH, newH)
  } while (!succ)

  /* copy input to ringbuffer */

  lwsync
  while (prod.tail != oldH);
  prod.tail = newH
}
```

図 3 リングバッファへのデータ追加の擬似コード．コードは DPDK の最新の lib/librte_ring/rte_ring.c と lib/librte_ring/rte_ring_generic.c (commit 05e0eee) を簡易化したものである．prod は producer 側の書き込みインデックスとして，head と tail の 2 つを値として持つ．comp_and_set はアトミック操作で，prod.head が oldH であれば newH へ prod.head を更新する．lwsync は POWER のメモリ同期命令である．

クフリーでない場合は開発者が意識しなくとも， pthread などのマルチスレッドの実装のためのライブラリがメモリ同期を管理してくれることが多い．ロックを使わないケースではその管理は開発者の責任となるため非決定的バグの原因となりやすい．実際 DPDK はすでにメモリ同期のプリミティブを提供し，ARM や POWER への対応をしているとされているにも関わらず，テストが不十分なのか移植の初期では非決定的なクラッシュを引き起こしていた．

図 3 はクラッシュの原因となっていた既存コードを簡易化したものである．このコードは，SPDK のイベントを管理するためのリングバッファの producer 側のファイルシステムイベントの追記に関するものである．x86 CPU は lwsync は必要ない一方で，POWER や ARM の場合，図中の prod.head や prod.tail の書き出しが他の CPU に反映されない．その結果，複数スレッドによるイベントの追記が起きた場合に，prod.head や prod.tail の読み出しに古いデータが使われる恐れがある．

図にあるように，既にいくつか lwsync が挿入されているものの，それらはメモリの読み出し順序が原因で起こる別のバグ修正のためのものであった．図 3 は省略されているものの，newH の計算に prod.tail が使われており，prod.head の直後の lwsync が抜けていたために，prod.head と prod.tail の値の読み出しが別順に起きていたものであった．その結果，producer-consumer の競合が起きていたものの，今回は producer-producer 間の競合は修正できていなかった．

```
WriteThread::AwaitState(...) {
  for (tries = 0; tries < 200; ++tries) {
    state = w->state.load(
         std::memory_order_acquire);
    if ((state & goal_mask) != 0) {
      return state;
    }
    port::AsmVolatilePause();
  }
  ....
```

図 4 メモリ同期の修正前のコード

3.3.2 リファレンスカウンタ

高い性能を目指すソフトウェアは明示的なメモリ管理を必要とする C++ などで実装されることが多い．そのため，確保したヒープメモリのリファレンスカウンタの管理が他のソフトウェアに比べて重要になる．スレッドセーフの実装にするために，アトミック変数を用いて実装される．

MongoDB は C++ の std::atomic を用いて BSON (JSON のバイナリ表現) オブジェクトのためのメモリバッファを管理している．MongoDB 以外にも RocksDB もデータ書き込みのスレッドの状態管理に利用している．std::atomic はアトミックな値のアクセスとともに，アクセスされた値に関連するメモリアクセスの順序づけについて細かなチューニングを可能にするものである．このチューニングは強いメモリモデルを取るインテル CPU にはほとんど関係のないものとなっているものの，POWER では性能に大きく影響を及ぼすものとなっている．

std::atomic がサポートするメモリオーダーは大きく分けて 3 つ存在する．順序を制約しないもの (memory_order_relaxed), 操作の前または後での順序制約を課すもの (memory_order_acquire, memory_order_release, memory_order_acq_rel), 操作の前後で全ての変更順序を保持するもの (memory_order_seq_cst) に分けられる．C++ のデフォルトでは全てのアトミック演算は明示的に指定しない限りは最も強い memory_order_seq_cst が利用される．

図 4 は移植での変更前の RocksDB のデータ挿入時に呼ばれるコードである (RocksDB 5.6.1 の db/write_thread.cc)．RocksDB はデータ挿入するときに x86 環境では最初に 200 回のポーズ命令を呼び，その間に insert リクエストをバッチするようになっている．コードに書かれたコメントによると，1 msec の遅延をとっているとされているものの，最適化前の POWER ではポーズの前にバッチステートを示すフラグに対して memory_order_acquire によるメモリロード (POWER の場合 isync) が入っており，そこでの遅延が想定よりも大きくなってしまっていると予想される．今回の移植では memory_order_relaxed を指定し，それと対になるフラグへのメモリストアを memory_order_relaxed

から memory_order_release (POWER の場合 lwsync) に変更することで，性能を向上させている．

3.4 SIMD 命令

SIMD 命令はデータのハッシュ値や CRC 値の計算の高速化に使われてるアーキテクチャ依存コードである．SPDK を呼び出す RocksDB 内でも x86 向けのコードは高速な SIMD 命令を使い，他のアーキテクチャでは低速な互換ループ処理を行うようになっている．SPDK がサポートしていない最新の RocksDB には POWER のための SIMD 命令が加えられているため，本研究では最新の RocksDB から当該部分をバックポートし，POWER の SIMD 命令を SPDK を有効にした RocksDB でも使えるようにする．

4. MongoDB の SPDK 有効化

MongoDB はストレージエンジンをクエリ処理エンジンと独立させ，モジュールとして入れ替えることができるようになっている．本研究では SPDK で既にベンチマークが提供されている，RocksDB 5.6.1 を MongoDB 3.4 のストレージエンジンとして動作させる．RocksDB そのものは LevelDB と同様にシンプルなデータストアのためのライブラリであり，スタンドアロンで動作することよりも，MongoDB などの他のデータベースのストレージエンジンとして動作することを想定されている．

そのため，現在 RocksDB と SPDK の連携するコードは RocksDB 単独のベンチマークコードのみが公開されており，別のレポジトリで提供されている MongoDB と RocksDB のバインド [9] を SPDK の API を呼ぶように改変する必要があった．その変更は 16 行のコード改変が必要となり，その他に新しい設定項目の追加などのために 56 行変更した．MongoDB および RocksDB は C++ の atomic 命令の使い方に関する変更が必要であり，MongoDB は 19 行，RocksDB は 3 行の変更を要した．SIMD 命令のバックポートは 1970 行の変更を要している．

また，既存の SPDK の Blobfs の API は MongoDB からの複数スレッドの呼び出しで問題が起こるようになるため，Blobfs の修正も必要となった．Blobfs ではイベントチャネルをスレッドローカルストレージに保持しており，異なるスレッドからの呼び出しごとにイベントチャネルの初期化が必要であった．最終的に，この変更のため SPDK は 31 行の追加が必要であった．

DPDK は IOMMU のマッピングに関する変更は 7 行要し，また MSI-X のチェックを削除するための 1 行の変更と 3 行の lwsync の呼び出しの追加が必要であった．Linux カーネルの VFIO に関する変更は IBM の OpenPOWER のコードレポジトリに公開されているものを利用した [10]．

5. 実験

本研究は移植した SPDK をバインドした MongoDB の性能を YCSB を用い，POWER とインテル環境それぞれで測定を行う．また，POWER において SPDK バインドした MongoDB と WiredTiger をバインドした MongoDB，SPDK なしで RoksDB をバインドした MongoDB, Redis との性能比較をする (本節ではそれぞれ SPDK, WiredTiger, RocksDB, Redis と呼ぶ)．本研究の全ての実験結果は 3 回繰り返した結果の平均としている．評価基準としてスループット (処理したリクエスト数 / 全体の実行時間) とレイテンシ (一回のリクエストが完了する時間) の平均を用いる．

5.1 使用するプラットフォーム

POWER プラットフォームとして，IBM Power System S822LC (8335-GCA) を利用する．CPU は 160 論理コアの POWER8 CPU (2 ソケット X 10 物理コア/ソケット X 8 論理コア/物理コア) を用いる．インテル環境には Lenovo System x3650 M5 を利用する．CPU は 72 論理コアの Intel Xeon CPU E5-2699 v3 (2 ソケット X 18 物理コア/ソケット X 2 論理コア/物理コア) を用いる．どちらの環境もシステム物理メモリは 512 GB, ストレージに HGST Ultrastar SN100 Series NVMe SSD を利用する．OS は Ubuntu 18.04LTS, VFIO を改変した Linux 4.10.0 を利用する．RocksDB, SPDK バインドした RocksDB は 127GB をデータベースキャッシュとして利用する設定とする．SPDK のキャッシュサイズは 10 GB とし，圧縮は全て無効にしている．その他の設定は全てデフォルトとした．

YCSB クライアントは 0.13.0 を利用し，クライアントは別の P8 マシンで動作させ，10 Gbps ネットワーク経由でデータを送受信する．クライアントとして利用するマシンはサーバと同じ IBM Power System S822LC (8335-GCA) を利用する．ただし，CPU は 152 論理コア (2 ソケット X 9 物理コア/ソケット X 8 論理コア/物理コア) で，512GB RAM を用いる．クライアントの OS は Ubuntu 18.04LTS, Linux 4.17.2 を用いる．

5.2 クライアントの設定

YCSB は NoSQL などの，シンプルなデータベースの read, scan, insert, update 操作に関する基本的な性能を測定することができる．デフォルトのワークロードとして workload A から F までのワークロードが準備されており，それぞれ異なる read, scan, insert, update 操作の割合を持ち，それぞれターゲットとなるワークロードが異なる．本研究はそのうち，workload A, B, C をワークロー

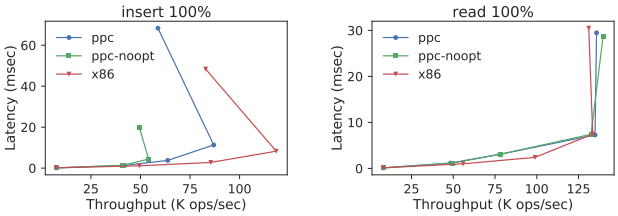

図 5 Haswell および P8 での SPDK の性能 (左: Load, 右: Workload C). ppc-noopt はリファレンスカウンタおよび SIMD の最適化前の POWER8 での実験結果を示す.

ドとして用いる．workload A は read と update の割合が 50%: 50% となるようにランダムで生成されるワークロードとなっており，ショッピングサイトのような最近のユーザセッションの操作を記録するようなケースを想定している．workload B は read が 95% に対して update が 5% と少なく，SNS の写真のタグ付けを主に想定している．workload C は read が 100% のもので，Hadoop など他のアプリケーションによって作られたユーザプロファイルをキャッシュするためにデータベースを使う想定をしている．

評価ではクライアント数を 1, 8, 16, 32, 64 と変更し，それに伴い読み書きするレコード数を 100 万, 800 万, 1600 万, 3200 万, 6400 万と変更して性能を測定する．レコードはそれぞれ YCSB のデフォルトである 100 バイトのフィールドを 10 個含み，おおよそレコードあたり 1 キロバイトの大きさとなるようにしている．

5.3 SPDK の Haswell および P8 での性能評価結果

図 5 は YCSB のデータロードおよび Workload C のスループット対レイテンシを示している．POWER8 での最適化の結果，データロードは最大でスループットが 75% 向上した．この結果は RocksDB のデータ書き込みのバッチ部分のメモリ同期命令の変更が大きく影響していた．また，最適化により POWER での性能は大きく改善したものの，Haswell のデータロードに比べると 26% スループットが低い．現状では原因は解明できていないものの，少なくとも CPU 利用率はそれほど高くないため，mutex などによるスレッドのスリープが多く起きてしまっていると予想している．

一方で，Workload C において POWER8 は Haswell 環境に比べてスループットは最適化前で 6%, 最適化後で 3% 改善していた．少なくともこの結果から，データ読み出しに関しては，移植後の POWER とインテル環境に大きな差が生じていないことがわかる．

図 6, 7 はそれぞれ Workload A と Workload B の結果を示している．Workload A のようにデータの更新が多くなると，Haswell との差が大きくなり (最大 20% のスループット低下)，Workload B のように読み出しが多い場合は差がほとんど見られない (最大 3% のスループット向上).

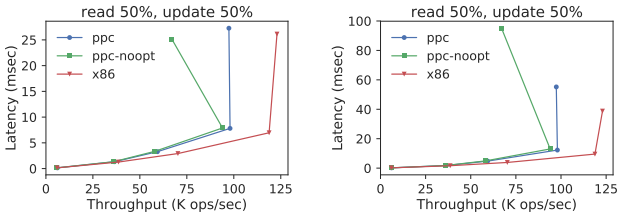

図 6 Haswell および P8 での SPDK の性能 (Workload A, 左: Read, 右: Update)

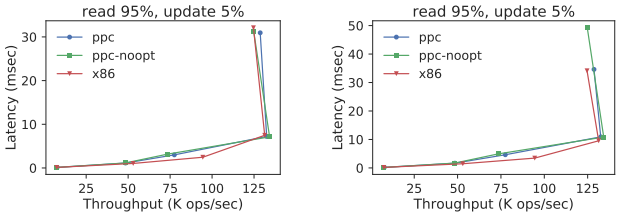

図 7 Haswell および P8 での SPDK の性能 (Workload B, 左: Read, 右: Update)

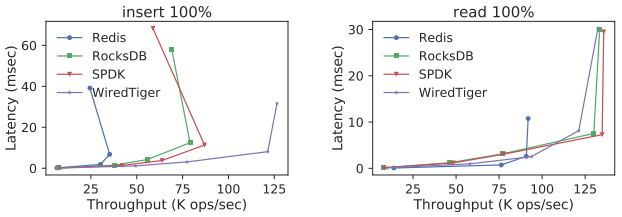

図 8 データベース性能比較 (左: Load, 右: Workload C)

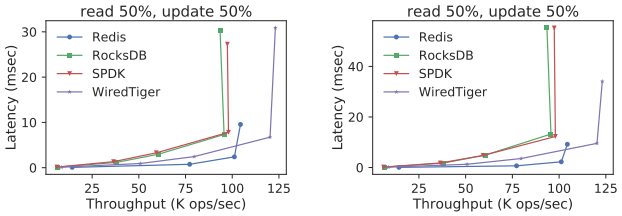

図 9 データベース性能比較 (Workload A, 左: Read, 右: Update)

読み出しの差が小さい原因として，YCSB のデフォルトでは読み出しするキーの分散に Zipfian (キーの偏りが現実のワークロードを反映して大きい) を利用しており，データの多くはデータベースのキャッシュメモリから読み出されるためと考えられる．Workload A もデータロードと同様に，最適化の効果により，性能限界を超えた後のスループットが最大 45% 向上し，不可増大時のスループットの大幅な減少が抑えられている．

5.4 他のデータベースとの比較結果

図 8, 9, 10 はそれぞれデータロード，Workload C, Workload A, Workload B の異なるデータベースでの測定結果を示している．特に，図の RocksDB と SPDK を比較することで，SPDK のデータベースワークロードにとってのアド

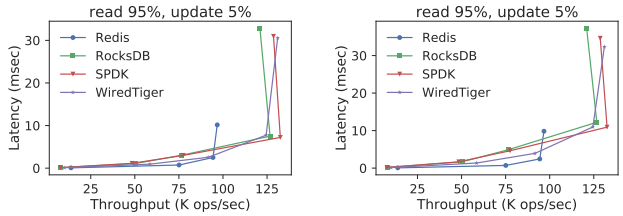

図 10 データベース性能比較 (Workload B，左: Read，右: Update)

バンテージを推測することができる．データロードの限界性能を超えた状況下を例外とすると，SPDK は RocksDB よりも良いスループット，レイテンシを示している．例えば図 10 は SPDK が RocksDB よりもスループットが最大 4%，レイテンシは 3 msec 改善している．

他の傾向として，Redis は低負荷時のレイテンシが最も良いものの，限界性能が他に比べて低い．Redis の性能限界の低さは 1 CPU のみ使うデザインになっているためであると予想されるため，プロセス数を増やすなどの方法を取れば他のデータベースの性能に近づいていくと予想される．WiredTiger は書き出し性能で SPDK よりも優位性を示している．WiredTiger は読み書き両方の性能を高めるために，Log-structured merge-tree (LSM tree) を最初に作り，その後に読み出し用の B-tree を作成することから，読み出し側の性能が上がりにくい側面もある．そのため，図 8 右や図 10 のように，限界性能に注目した場合，SPDK や RocksDB に読み出しの優位性が発生している．一方で，SPDK の Haswell での結果と比べると，書き出し側の WiredTiger の大きな性能アドバンテージは RocksDB とのデータベースレベルの差異も無視できないものとなっていると考えられる．

6. 関連研究

本研究は特に仮想化機構および仮想メモリに関するポーティングのひとつの研究とみなせる．近年の仮想化機能に関するポーティングは，ARM システムの移植が主なトピックとして知られている ([1] など)．ARM での nested virtualization の適用は，エミュレーションベースの仮想化 [11] は ARM では性能が出せないことを示している．

POWER においては，IBM の所有するプロダクトの多くが依存している Java ランタイムや Java ワークロードの最適化がされてきている．例えば OpenJDK のガベージコレクションの弱いメモリモデルに関連した最適化がされている [12]．POWER 特有の最適化のその他の例として，Spark をワークロードとした，物理コアあたりの論理コア数の調整に関する研究がされている [13]．Java virtual machine や Spark も SPDK と同様に，インテル互換環境が主な想定利用環境とされてきた経緯から，いずれもアーキテクチャ固有の問題に関する話題となりやすい．

本研究の対象とする，ユーザレベルデバイスドライバは性能または信頼性の観点から，有用性が模索されている．性能の観点では，ユーザレベルデバイスドライバはコンテキストスイッチのオーバヘッドを減らすことや，ダブルバッファリングなどの問題を解決する方法として知られている [2]．SUD はカーネルレベルデバイスドライバをユーザ空間に移植し，DMA や割り込みによる攻撃を防ぐために仮想化機構を利用している [14]．

7. まとめと今後の課題

本研究は POWER システムへ SPDK を移植・最適化し，実験結果を示した．最適化は主にデータ書き出しに関して効果のあるもので，最大 76% のスループット向上が見られた．しかし，既存の Haswell システムでの実験結果と比べると，移植後の SPDK はいまだ大きな性能差を引き起こしていた．今後はデータ更新についての最適化を進めていくことで，POWER システムで SPDK を活用できる可能性を増やしていく必要がある．

参考文献

[1] Lim, J. T., Dall, C., Li, S.-W., Nieh, J. and Zyngier, M.: NEVE: Nested Virtualization Extensions for ARM, *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 201–217 (2017).
[2] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T. and Roscoe, T.: Arrakis: The Operating System Is the Control Plane, *ACM Transactions on Computer Systems (TOCS)*, Vol. 33, No. 4, pp. 11:1–11:30 (2015).
[3] : Storage Performance Development Kit, http://www.spdk.io.
[4] 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)*, pp. 143–154 (2010).
[5] : DPDK, http://www.dpdk.org.
[6] : SPDK:Blobfs (Blobstore Filesystem), http://www.spdk.io/doc/blobfs.html.
[7] : VFIO - "Virtual Function I/O", https://www.kernel.org/doc/Documentation/vfio.txt.
[8] Luc Maranget, S. S. and Sewell, P.: A tutorial introduction to the ARM and POWER relaxed memory models, http://www.cl.cam.ac.uk/ pes20/ppc-supplemental/test7.pdf.
[9] : MongoDB storage integration layer for the Rocks storage engine, https://github.com/mongodb-partners/mongo-rocks.
[10] : Linux kernel source tree, https://github.com/open-power-host-os/linux/tree/hostos-stable.
[11] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O. and Yassour, B.-A.: The Turtles Project: Design and Implementation of Nested Virtualization, *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, pp. 423–436 (2010).
[12] Horie, M., Horii, H., Ogata, K. and Onodera, T.: Balanced Double Queues for GC Work-stealing on Weak

Memory Models, *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM '18)*, pp. 109–119 (2018).

[13] Jia, Z., Xue, C., Chen, G., Zhan, J., Zhang, L., Lin, Y. and Hofstee, P.: Auto-tuning Spark Big Data Workloads on POWER8: Prediction-Based Dynamic SMT Threading, *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*, pp. 387–400 (2016).

[14] Boyd-Wickizer, S. and Zeldovich, N.: Tolerating Malicious Device Drivers in Linux, *Proceedings of the 2010 USENIX Conference on Annual Technical Conference (USENIXATC '10)* (2010).