追記型インターフェイスによる SSDのテイルレイテンシ改善

田所 秀和[1]　山口 健作[1]　長谷川 揚平[1]

概要：SSD を利用したシステムでは、ガーベジコレクション (GC) を原因としたテイルレイテンシが報告されている。これは、ブロック単位での消去しかできない NAND 型フラッシュメモリを、セクター単位などのより細かい粒度で利用することが原因である。また、大きなシステムではテイルレイテンシは増幅し問題になりやすいことが知られている。さらに、他のユーザのリクエストによって発生した GC の影響を受けてしまい、一定のサービスレベルを維持できない可能性もある。このような問題を解決するために、大規模ストレージシステムを想定した GC が発生しない SSD を提案する。この SSD は、大規模ストレージシステムが追記型データ構造で構成されていることに着目し、基本的にシーケンシャル書き込みのみを許容する。このような制約により、ブロック内のデータが断片化せず、GC が不要である。また、シーケンシャル書き込みの制約を緩和する buffer pinning と呼ばれる機能を持っている。これは、SSD 内の DRAM を利用することで、一部の上書きを可能にする。この SSD をソフトウェアエミュレータとして実装し、その上に log structured merge tree を利用する RocksDB を移植し動作することを確認した。実験では、GC が発生する通常の SSD と比較して、RocksDB における読み込みの 99.99%ile レイテンシが約 1/8 に低減できていることを確認した。

Hidekazu Tadokoro[1]　Kensaku Yamaguchi[1]　Yohei Hasegawa[1]

1. はじめに

NAND 型フラッシュメモリを使った Solid State Drive (SSD) が広く使われている。従来の HDD と比較して、高性能、省電力、可動部がなく衝撃に強い、形状も比較的自由で小型化可能などの理由で、HDD を置き換えつつある。NAND 型フラッシュメモリは、*page* 単位の読み書きと、page よりも大きな *block* と呼ばれる単位でのみ消去が可能である。また、一度書いた page は消去することにより再度利用できる。そのため、SSD は定期的にガーベジコレクション (GC) と呼ばれる操作をする必要がある。GC では、block 内の有効な page を別の block へコピーし、古い block を消去する。こうすることで、無効な page を再び利用することができる。

SSD の GC は、レイテンシ発生の要因となっている。文献 [1] では、RAID グループにおいてその 0.58%がレイテンシの中央値の 2 倍を越えており、実際に SSD における遅延の発生が観測されている。また、大規模分散型システムでは、レイテンシの影響が増幅し、より多くのユーザに影響を与える可能性が指摘されている [2]。多数のノードで 1 つのリクエストを処理する場合、1 ノードの処理が遅れると全体の処理が遅れてしまう。その結果、Web サイトの応答が遅いなど、ユーザ離れにつながる恐れがある。サービス水準合意を保証する際の障害との指摘もある [3]。

このような問題を解決するために、GC が原理的に発生しない SSD を提案する。この SSD は、*segment* と呼ばれる単位で SSD を管理する。ユーザは、segment の先頭から順番にのみ書き込みを行うことができる。この SSD は、segment 内のどこまで書かれたかを管理しており、一度書き込んだ場所や、遠く離れた場所に書きこもうとするとエラーになる。このため、書き込みの単位は sector 単位であるが、必ずシーケンシャルに書き込む必要がある。segment はフラッシュメモリの消去の単位である block の整数倍であり、ユーザはこの segment の単位で消去を行う。消去された segment は、再び先頭から書き始めることができる。読み出しは、sector 単位でランダムに行うことができる。SSD にこのような制限を加えることで、block 内で断片化が発生せず、GC が不要になる。

この提案 SSD は、広く使われている追記型のデータ構造

[1] 東芝メモリ株式会社 メモリ技術研究所

を利用するストレージシステムをターゲットとして考案した。追記型ストレージの一つである log-structured filesystem は、変更の差分をログとしてストレージデバイスに追記していく [4, 5]。これにより、ストレージデバイスの高速なシーケンシャル書き込み性能を生かすことができる。Key-value store の一種である LevelDB [6] や RocksDB [7] は、log-structured merge tree (LSM tree) [8] と呼ばれる追記型のデータ構造を利用している。また、大規模ストレージの一つである Windows Azure Storage [9] では、一貫性の確保のために Stream という追記のみのデータ構造を利用している。Google Bigtable [10] や Apache Cassandra [11] は LSM tree をベースとしており、SSTable と呼ばれるファイルへ追記していく。また、データベースにおける基本的な技術である Multi Version Concurrency Control [12] や Write Ahead Logging [13] も追記を基本としている。

我々は、この SSD をソフトウェアエミュレータ上に実装した。この SSD エミュレータは、NAND 型フラッシュメモリの遅延やチャネル数などのパラメータを取り、実時間で応答する。そのため、SSD 内部の挙動をシミュレートするだけのソフトウェアとは異なり、実際にアプリケーションを動かし性能を評価することができる。このエミュレータ上に、LSM tree をデータ構造として利用する RocksDB を移植し、追記のみの SSD でも動作可能であることを確認した。

実験では、実 SSD と提案 SSD との比較を行った。エミュレータのパラメータを実 SSD とできる限り同じ性能になるように設定した。このパラメータの決定のために、実際に性能を測定したり、同じ世代の NAND 型フラッシュメモリのパラメータを参考にした。実 SSD で GC が発生する状態では RocksDB の性能が劣化しテイルレイテンシが発生するが、提案 SSD ではそのようなことは起きないことが確認できた。

本論文の構成を以下に示す。2 章では、研究の背景となる SSD とストレージシステムについて述べる。3 章では提案する SSD について、4 章ではそのエミュレータ実装とアプリケーションの移植について述べる。5 章では、実験結果およびその考察について述べる。6 章では関連する研究を紹介し、7 章でまとめる。

2. 背景

SSD は、NAND 型フラッシュメモリなどの半導体メモリを利用したストレージデバイスである。HDD などと互換性を持たせるため、512B や 4KB などの sector 単位で読み書きができる。一方で、NAND 型フラッシュメモリは sector よりも大きく 16KB ほどの page と呼ばれる単位でしか読み書きしかできない。また、一度書き込まれた page は消去してから再利用しなければならないが、消去は page よりも大きな block 単位で行う必要がある。この block は page の約 1000 倍ほどの大きさになってしまう。SSD の内部は、図 1 にあるように、controller から channel を通して複数の plane が繋っており、その中に消去の単位である block と、その要素である page が存在する。

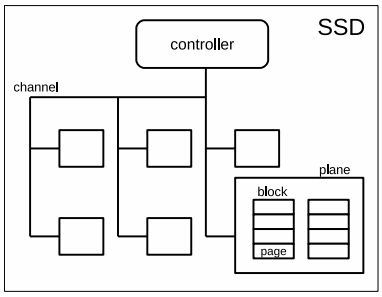

図 1　SSD の概要

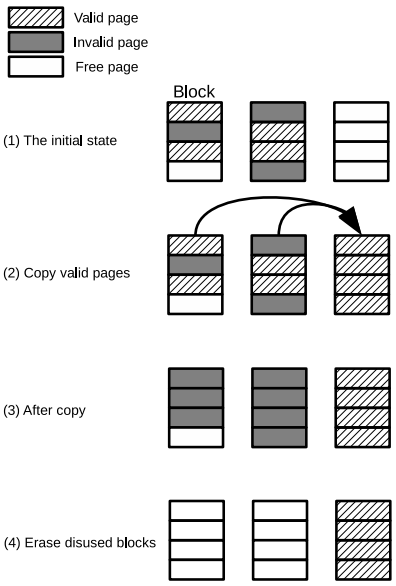

図 2　GC の 4 ステップ

Page には 3 種類の状態が存在することになる。データが書き込まれた *valid* 領域、書き込まれてない *free* 領域、そして論理的には消去したが物理的には消去されていない *invalid* 領域である。データを書き込む領域を確保するために、新たに free page のみの block を作る必要がある。そのために、valid page を集めて別の block に移動させ、invalid page だけの block を作り出し、その block を消去する。図 2 は、3 つの block が存在し、GC によって新たに 1 つの free block を作り出す様子である。初期状態 (1) では、2 つの block が使われており、その block には invalid page が含まれている。そこで、(2) で free block に valid page をコピーする。その結果、(3) のように 2 つの block を valid な page が存在しないようにできる。最後に (4) で不要な block を消去することで、新たな free block を作り出すことができる。

GC を処理中の SSD では、通常では発生しない追加のレイテンシが発生する場合がある。Controller が内部的なデータの読み書きのために、channel や plane などの共有

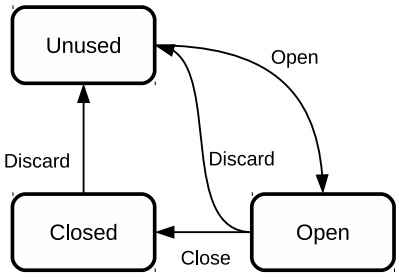

図 3　segment の状態遷移

リソースを使うからである。例えば、図 2 の (2) の状態で、valid page を読む必要がある場合、コピーが完了してから読む必要がある。また、(4) の erase 中では、同じ plane 内を読むことはできず、erase の完了を待つ必要がある。

さらに、GC におけるレイテンシの増大とは別の問題として、書き込み増幅と呼ばれる現象が発生する。図 2 では、valid page を移すため、ホストの IO とは独立した IO が発生している。ホストからの IO と GC を含めた SSD 全体の IO との比率を、write amplification factor (WAF) と呼ぶ。WAF は SSD の寿命を決める重要な要素であり、低いほど良い。

3. 提案 SSD

GC に起因するレイテンシや書き込み増幅の問題を解決するため、原理的に GC が発生しない SSD を提案する。この SSD は、追記型の書き込みのみを許し、ランダムな書き込みを禁止する。これにより、block 内に invalid な page が発生せず、valid page を集めるための GC が不要になる。読み込みについては、従来の SSD と同じく sector 単位でのランダム読み込みをサポートする。

3.1 Segment Based IO

この SSD は、block の整数倍となる *segment* と呼ばれる単位を IO の基本とする。segment は状態を持っており、図 3 のような状態を遷移する。最初の状態は unused であり、データを保持していない。ホストが書き込みを行う場合には、書き込みたい segment を *open* することでその segment が書き込み可能になる。Open 状態の segment に対して、sector 単位で先頭から順番に一度だけ書くことができる。いきなり segment の途中から書くことはできず、また、一度書き込んだ sector に再び書くことはできない。そのような書き込みは失敗する。最後まで書き切った segment と、一度 *close* した segment は、それ以上追記することはできない。このような segment を再び書き込み可能にするには、一度 *discard* する必要がある。

提案 SSD は、従来の SSD と同様に sector に LBA と呼ばれる番号がついている。図 4 にあるように、segment は複数の sector から構成されており、各 segment は同じ数の sector を含んでいる。Segment 内の sector 数を segment size、

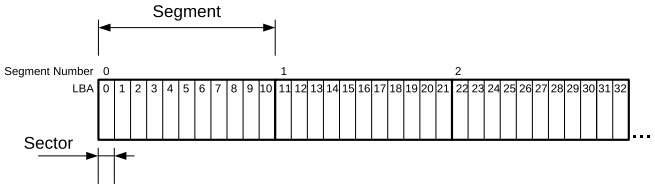

図 4　11 の sector から構成される segment の例

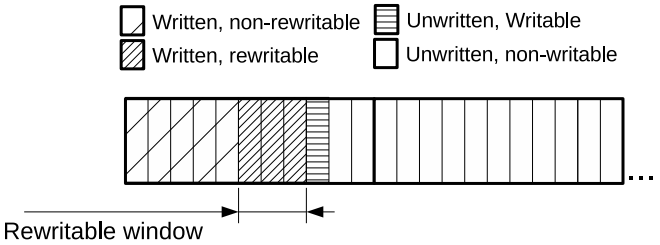

図 5　buffer pinning による 3 sector の rewritable window

segment の先頭 sector の LBA を segment head LBA とすると、*segment number*∗*segment size* = *segment head LBA* の関係にある。

3.2 Buffer Pinning

前節で述べたように、提案 SSD では、基本的に segment 内では sector 単位で追記のみを許す。しかし、このような制限ではランダム書き込みが必要な用途には使えず、応用が限られてしまう。そこで、追記の制限を緩める buffer pinning と呼ばれる機能を提供している。Buffer pinning は、*rewritable window* と呼ばれる書き換え可能な範囲を提供する。Segment を open する際に rewritable window N を指定すると、最近書いた N sector が書き換え可能な状態で管理される。図 5 は、rewritable window $N = 3$ を指定し、最近書き込まれた 3 つの sector が書き換え可能な状態にある例である。

3.3 冗長性

提案 SSD は、冗長性や並列度をユーザが指定できる構造になっている。$M + K$ の消失訂正符号を提供し、M と K をユーザが指定する。例えば、ストライピングのサイズである M を大きくすることで、SSD 内部の並列度を生かし、IO 性能を向上させることができる。このとき SSD 内部では、segment が M block から構成され、segment は block の M 倍になる。また、K を大きくすることで、データの消失により強くなる。

3.4 API

提案 SSD はユーザが使いやすいように、API を提供している。図 6 は、API の簡単な使い方を示している。この例では、先頭の segment を開き、4KB 単位で 5 回追記を行っている。

```
1  void append_20KB(void) {
2    struct rad_device r;
3    rad_device_open(&r, "/dev/rad0n1s1");
4
5    struct rad_segment seg;
6    const int seg_n = 0; /* 0 番目の segment */
7    rad_segment_init(&r, &seg, seg_n);
8    rad_segment_update(&seg);
9
10   /* segment がopen or closed なら消去*/
11   if (rad_segment_is_open(&seg)
12         || rad_segment_is_closed(&seg)) {
13     rad_segment_discard(&seg);
14   }
15   assert(rad_segment_is_unused(&seg));
16
17   /* 消去したので先頭から書ける*/
18   rad_segment_open(&seg);
19
20   /* 4 KB のaligned バッファを確保*/
21   char* buf = aligned_alloc(4096, 4096);
22   /* zero で書き込む*/
23   memset(buf, 0, 4096);
24
25   /* 4 KB を5 回追記*/
26   for (int i = 0; i < 5; i++) {
27     rad_segment_append(seg, buf, count);
28   }
29
30   free(buf);
31
32   rad_segment_close(&seg);
33   rad_device_close(&r);
34 }
```

図 6　API を使い先頭から 20KB をゼロで埋める例

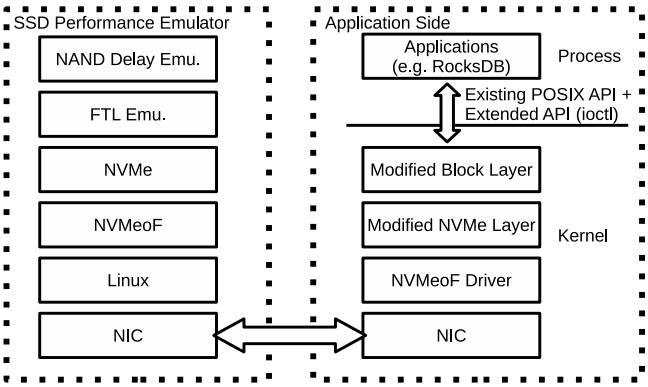

図 7　SSD エミュレータの概要

4. 実装

4.1 SSD エミュレータ

提案 SSD の動作をエミュレートするソフトウェアを作成した。この SSD エミュレータは、NVMe コマンドを受け取りその結果を返すことで、SSD のように振る舞う。図 7 が、その概要である。このエミュレータは、性能を模倣するために CPU を積極的に利用し、また、データを保存する先として主記憶を大量に利用する。エミュレータとクライアントは異なるマシンで動作し、その間を NVMeoF によって接続することで、NVMe コマンドをやりとりする。異なるマシンで動作させる理由は、性能の干渉を抑制するためや、データを保存するための主記憶が大量に必要なためである。

このエミュレータは、SSD モデルに従い遅延をエミュレートすることで、SSD の機能だけでなく性能も模倣する。SSD モデルは NAND 型フラッシュメモリ、NVMe フロントエンド、Flash Translation Layer (FTL) から構成される。コントローラは、チャネルを通し Open NAND Flash Interface (ONFI) 4.0 [14] に従って NAND 型フラッシュメモリへのコマンドを発行する。NAND 型フラッシュメモリのエミュレーションのために、各コマンドに対する遅延が定義されている。最終的な遅延は、NAND 型フラッシュメモリのコマンドの遅延と、その NAND 型フラッシュメモリにアクセスするためのチャネルの占有状態と転送データ量で決まる。例えば、同じチャネルを使うコマンドは並列には発行できず、その前のコマンドの発行の完了を待つ必要がある。

デバイス構成のパラメータとして、plane 数、ブロック数、ページ数、1 つのセルに何ビットの情報を保持するか (SLC・MLC・TLC など) がある。NAND 型フラッシュメモリの処理時間のパラメータは、読み書きの時間である tR と tPROG が存在し、消去の時間 tBERS などがある。エラーに関しては現在は実装しておらず、NAND 型への読み書きはビット化けなどの物理的なエラーは発生しない。FTL は、通常の SSD の sector 単位とは異なり、block 単位で管理する。これは、提案 SSD が block 単位の整数倍である segment 単位で書き込みと消去を行うためである。

エミュレータは、Intel SPDK [15] を拡張することで実装した。SPDK は高性能なストレージアプリケーションを記述するためのソフトウェアである。ユーザランドで動作することや、ポーリングによってハードウェアにアクセスし、ロックを使わないプログラミングモデルなどの特徴によって、高性能を実現している。エミュレータでは、SPDK の提供する機能のうち、NVMeoF、NVMe、DRAM ベースのブロックデバイスを利用した。NVMe レイヤを拡張し、提案 SSD 用の NVMe コマンドを実装している。この NVMe レイヤから DRAM ベースのブロックデバイスへアクセスする間に、SSD モデルに基づく遅延を実現する層を入れた。

クライアント側は、Linux Kernel の NVMe ドライバを拡張することで、提案 SSD のインターフェイスに対応させた。提案 SSD は、Read や Write などの通常の NVMe の

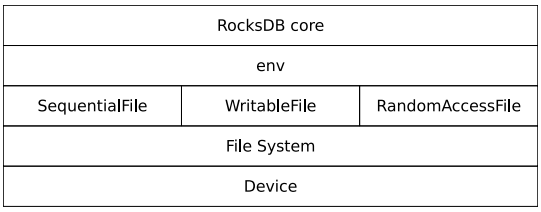

図 8　RocksDB の構成

コマンドに加えて、segment の操作など NVMe 標準に存在しない拡張をサポートしている。このため、既存の NVMe ドライバにこれらの拡張をサポートするように実装した。segment の操作などはアプリケーションからは ioctl を使いドライバに指示を出すようにしている。一方、書き込みや読み込みなどは通常の read/write システムコールを利用する。3.4 節で述べた API は、これらの read/write や ioctl を抽象化したユーザランドライブラリとして提供している。

4.2 RocksDB

この SSD の API を使い動作するアプリケーションとして、RocksDB [7] を移植した。RocksDB は Key Value Store (KVS) と呼ばれるアプリケーションの一つである。データキャッシュとして使われるだけでなく、データベースのストレージエンジンなど、永続的にデータを保存する手段としても使われる。RocksDB は、log strucutred-merge tree (LSM tree) と呼ばれるデータ構造に基いている。LSM tree はデータを階層的に管理する。書き込み時には、データをファイルへ追記していき、一定量たまったら同じ階層でマージして下の階層へ移動させるという作業を繰替えす。この特性により、書き込みはシーケンシャルアクセスのみで実現でき、追記型ストレージに適している。

RocksDB は図 8 のように環境を抽象化した `Env` と呼ばれるクラスを通して、IO を発行する。これにより、環境に応じて `Env` を差し替えることで、さまざまな環境で RocksDB を動作させることが可能になる。`Env` は、RocksDB に必要なファイル操作を実装する必要がある。ファイル操作は、`SequentialFile`, `RandomAccessFile`, `WritableFile` というクラスで抽象化されており、これらを実装することで、RocksDB のファイル操作を新たに定義できる。`SequentialFile` は、ファイルに対してシーケンシャル読み込みを実現するクラスである。LSM tree をマージする時など、ファイルを一度に連続して読む必要がある場合に使われる。`RandomAccessFile` は、ファイルをランダムに読み込むためのクラスである。特定の Key に対する Get など、小さな単位で読み込むときに使われる。`WritableFile` は、ファイルへの追記を実現するクラスである。LSM tree のマージ時など追記が必要なときに使われる。

4.2.1 追記型ファイルシステム

提案 SSD 上で動く追記型のファイルシステムを作成した。この追記型ファイルシステムの目的は、RocksDB が必要とする最低限のファイル操作を提供し、RocksDB を提案 SSD 上で動かせるようにすることである。このファイルシステムは、一般的なファイルシステムとは違い、RocksDB に特化した使い方に制限している。このファイルシステムは、ユーザランドで動作し、提案 SSD の API を使いデータの読み書きを行う。RocksDB と静的にリンクされ、単一のアプリケーションとして SSD にアクセスする。

このファイルシステムは、ファイルへのランダム書き込みをサポートしない。4.2 節でも述べたように、RocksDB を動かすために必要なファイルの操作は、ランダム読み込み、シーケンシャル読み込み、追記のみである。これらのファイル操作を利用し、`SequentialFile`、`RandomAccessFile`、`WritableFile` を実装した。

追記方法は、*normal append* と *small append* の 2 種類を提供している。Normal append は sector 単位のみで追記を許可する。一度追記した場所は上書きできない。Small append は、sector よりも小さな単位で追記を実現する。Small append は、buffer pinning の機能と連携し、最後のセクター単位に満たない部分を上書き可能な状態で維持する。`WritableFile` がバイト単位でファイルシステムに追記したい場合、最後の sector 単位に満たない部分を `WritableFile` 側で保持しておき、追記したい部分を含めて sector 単位で上書きすることでバイト単位の追記を実現する。

RocksDB では、これらの 2 種類の追記を使い分けてファイルを書き込む。LSM tree を実現する SST ファイルの追記では、コンパクション時にファイルを書ききってしまうため、normal append で実現できる。WAL 用のファイルは細かい追記が発生するため、small append を利用する。このように、提案 SSD の限定された書き込み方法でも、RocksDB を動かすことが可能であることがわかった。

ファイルシステムのメタデータの領域は、すべて buffer pinning することによって、ランダムに書き込みができるようにしている。例えばファイルの追記があった場合など、データは追記すれば良いが、ファイルサイズなどのメタデータはデータ構造の途中を書き換える必要があり、Buffer Pinning によって対応した。

5. 実験

提案 SSD を実装した SSD エミュレータを用いて実験を行った。環境は、2 台のサーバマシンを InfiniBand スイッチを介して繋げた。サーバマシンは、CPU E5-2687W 3.10GHz CPU 2 socket * 10 core、250GB のメモリ、InfiniBand NIC は Mellanox MT27500 を用いた。比較対象のため、GC が発生する従来の SSD として、Intel 750 SSD を片方のマシンに接続した。

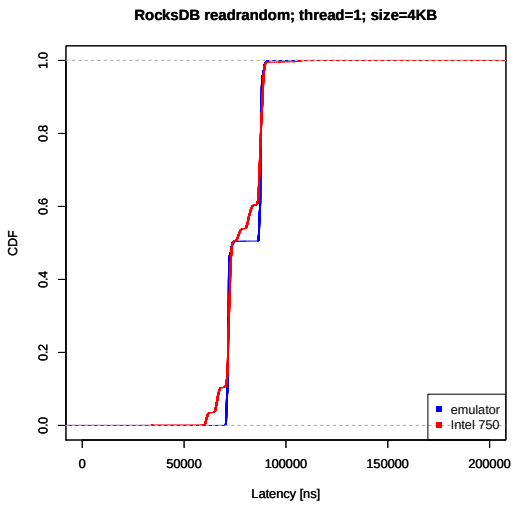

図 9　4KB リードのレイテンシの分布

表 1　SSD エミュレータのパラメータ

パラメータ名	パラメータ
チャネル数	14
LUN 数	2
tBERS	5000 μs
tR_lower	39 μs
tR_upper	55 μs
tPROG	1000 μs

5.1　SSD エミュレーション性能

GC が発生する従来の SSD と比較するため、エミュレータの性能パラメータを決定した。比較対象の SSD として、Intel 750 を用いた。表 5.1 が設定したパラメータである。リード性能は、実測によって性能の分布が一致するように設定した。これは、RDMA などの実装オーバーヘッドによる遅延が NAND 型フラッシュメモリのパラメータの tR に近いため、単に tR の値を設定しただけでは正しく性能をエミュレートできないためである。そのため、blktrace コマンドを用いてブロックレイヤでの 4KB リード性能が一致するように tR を設定した。RocksDB 付属の db_bench を利用し、readrandom における 4KB リードの遅延の分布が一致するようにした。図 9 がその結果である。大きく 2 つの段差があるのは、MLC における upper と lower の tR の速度差である。Intel 750 のほうが少し性能が良い部分が存在するのは、SSD の実装の違いであると考えられる。

それ以外のライト性能などのパラメータに関しては、該当 SSD で使われているのと同じ 20nm 世代のデバイスの典型的な値を用いた。書き込み性能に重要な NAND 型フラッシュメモリの tPROG の値は、RDMA などのオーバーヘッドと比較して十分に大きいため、エミュレーションに問題ないと考えられる。

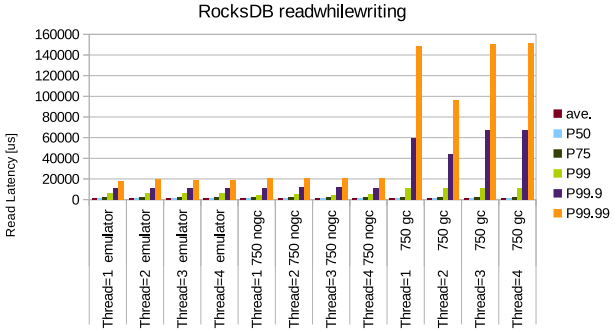

図 10　readwhilewriting でのリードレイテンシ

5.2　RocksDB

RocksDB を動かしたとき、GC が読み込みのレイテンシに影響を与えるかを調べる実験を行った。RocksDB 付属のベンチマークツール db_bench を用い、書き込みが同時に発生するワークロードである `readwhilewriting` と `readrandomwriterandom` で評価した。`readwhilewriting` は、バックグラウンドで 1 つのスレッドが常に書き込みを発生させる。`readrandomwriterandom` では、各スレッドが 9 対 1 の割合で読み込みと書き込みを発生させる。データサイズは 4KB を用い、スレッド数を 1 から 4 まで変化させた。比較対象の SSD は、GC が発生する状況と GC が発生しない場合の 2 通りを調べた。SSD に GC を発生させるパターンでは、各実験の前に 4KB のランダム書き込みを 8 時間実行し、書き込み性能が定常状態になるようにした。

図 10 が `readwhilewriting` における読み込みの平均レイテンシと 50%ile、99%ile、99.9%ile、99.99%ile レイテンシである。提案 SSD を実装したエミュレータと GC が発生しない状態の SSD では、レイテンシが小さく抑えられていることがわかる。一方で、GC が発生する場合には、リードレイテンシが大きくなっていることがわかる。GC が発生する SSD と比較して、提案 SSD のエミュレータでは、スレッド数 4 での 99.99%ile レイテンシにおいて、約 1/8 に抑えられている。

また、このときのテイルレイテンシの累積分布の全体を図 11 に、テイル部分の拡大を図 12 に示す。テイル部分を拡大したグラフに着目すると、提案 SSD のエミュレータでは、GC が発生しない状態の SSD と同じテイルレイテンシを達成していることがわかる。一方で、全体ではわずかに実 SSD のほうが性能がよい。図 9 で調べた通り、リード性能は一部で実 SSD のほうが優れている。これらはリードキャッシュのような、SSD 内部の実装の違いに起因すると考えられる。

図 13 が `readrandomwriterandom` における読み込みのレイテンシである。こちらの場合も、GC が発生する SSD と比較して、提案 SSD のレイテンシが小さく抑えられている。

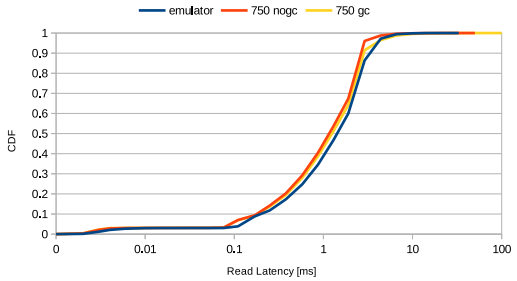

図 11　readwhilewriting: スレッド数 4 のときのリードレイテンシの全体の分布

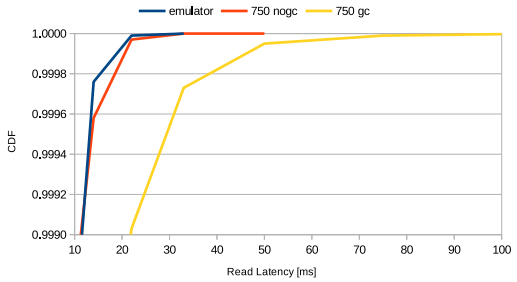

図 12　readwhilewriting: スレッド数 4 のときのリードのテイルレイテンシの分布

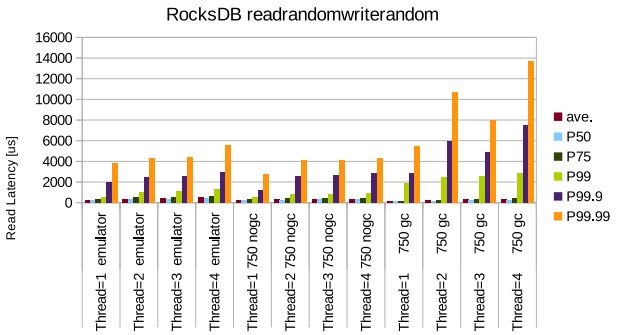

図 13　readrandomwriterandom でのリードレイテンシ

`readwhilewriting` と比較して、`readrandomwriterandom` でのテイルレイテンシ差が小さいのは、各スレッドが読み込みと書き込みを一定の割合で発生させるためである。`readwhilewriting` では、1 つのスレッドが常に書き込みを行うため、GC の発生頻度が多いと考えられる。

6. 関連研究

Software-Defined Flash [16] は、ソフトウェアから直接 Flash を制御するストレージである。大きな単位でシーケンシャルに書き込み、書き換えが無いという前提で、block 単位で書き込みを行う。GC が発生しない点や LSM tree のような追記型のストレージに特化しているという点で、提案 SSD と同じである。一方で、チャネルを直接ホストに見せ ioctl で read/write/erase を指示する特殊なインターフェイスになっている。提案 SSD では、互換性のため、NVMe を拡張したコマンドセットを提供し、Linux の block layer を経由して読み書きを実現している。

LightNVM [17] は open-channel SSD を Linux で使うためのサブシステムである。Open-Channel SSD はチャネルやチップといった SSD 内部の要素をホストから詳細に制御することで、性能向上や遅延を抑えることを意図している。

Application-managed flash [18] は、新たな IO インターフェイスを持つフラッシュストレージである。提案 SSD とほぼ同じく、セグメントという単位で IO を管理している。セグメント内では書き込みは追記のみ、読み込みはランダムに可能で、消去はセグメント全体を対象としている。任意の数のセグメントに対する書き込みが可能だと主張している。一方で、提案 SSD では segment を明示的に open する API を提供している。この open はライトバッファなどの SSD 内部のリソースの確保を同時に行っており、同時に open 可能な数は有限である。リソース管理する上で、任意の数のセグメントに対する書き込みが可能であるというのは、現実的ではないと我々は考えている。

FlashBlox [19] は、性能分離とウェアレベリングを同時に達成する仮想的な SSD を提案している。ダイ、チャネル、ブロックなど、さまざまな分離レベルを提供しつつ、それぞれに適したウェアレベリングを提供している。ブロック単位での消去とブロック内での追記を許し、API を提供する点で、提案 SSD と同じである。一方で、FlashBlox の目的は性能分離であり、評価もその観点でしか行っていない。テイルレイテンシの改善を目的とする提案 SSD とは異なる。

SSD の挙動を模倣するシミュレータ [20–22] が提案されている。これらは、ワークロードトレースを外部から与えることで動作する。SSD 内部の挙動を正確に知ることを目的にしており、実際にアプリケーションを動かすことは意図していない。VSSIM [23] は、QEMU をベースに作られた SSD エミュレータであり、OS をインストールすることで、ホストから本物の SSD のように扱うことができる。VSSIM は素朴な遅延モデルを採用しており、channel や plane の衝突など、SSD 内部の正確なモデル化をしていない。提案エミュレータは、ONFI に沿った NAND interface をモデル化しており、channel や plane の衝突による遅延もエミュレートする。

7. まとめ

書き込みを追記型に制限することで、原理的に GC を発生させない SSD を提案した。GC が発生しないことにより、テイルレイテンシの発生を抑えることができる。buffer pinning と呼ばれる SSD 内の DRAM を明示的に利用する API を使うことで、RocksDB のような LSM tree を使う KVS を動作させられることを確認した。性能を模倣するソフトウェアエミュレータを開発した。NAND 型フラッシュ

メモリやチャネルなどの SSD 内部をモデル化することで、実際に SSD を作成することなく、提案 SSD をアプリケーション上で評価することができた。実験では、RocksDB 付属のベンチマークツールを用いて、提案 SSD が GC の発生する SSD と比較して、99.99%ile テイルレイテンシを約 1/8 に低減できていることを確認した。

今後の課題は、より詳細な SSD モデルを作成することである。現在は、NAND 型フラッシュメモリの物理的なエラーを実装しておらず、エラー訂正による遅延などが評価できない。通常の SSD の FTL を作成することで、より汎用的な SSD エミュレータとして活用することもできる。また、評価に関しては、今回移植した RocksDB のような特定のアプリケーションだけでなく、F2FS などの追記型の汎用ファイルシステムを移植しさまざまアプリケーションを動かすことも必要である。

参考文献

[1] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, Haryadi S. Gunawi: The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments, *Proc. 14th USENIX Conf. File and Storage Technologies*, FAST'16.

[2] Dean, J. and Barroso, L. A: The Tail at Scale, *Commun. ACM*, Vol. 56, No. 2, pp. 74–80.

[3] Jaeho Kim, Donghee Lee, and Sam H. Noh: Towards SLO Complying SSDs Through OPS Isolation, *Proc. 13th USENIX Conf. File and Storage Technologies*, FAST'15.

[4] Mendel Rosenblum and John K. Ousterhou: The Design and Implementation of a Log-Structured File System, *Proc. 13th ACM Symp. Operating Systems Principles*, SOSP'91.

[5] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho: F2FS: A New File System for Flash Storage, *Proc. 13th USENIX Symp. File and Storage Technologies*, FAST '15.

[6] Sanjay Ghemawat and Jeff Dea: LevelDB, `http://code.google.com/p/leveldb`.

[7] Facebook: RocksDB, `http://rocksdb.org/`.

[8] Patrick O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth O'Neil: The log-structured merge-tree (LSM-tree), *Acta Inf.*, Vol. 33, No. 2, pp. 351–358 (1996).

[9] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas: Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency, *Proc. 23th ACM Symp. Operating Systems Principles*, SOSP'11.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber: Bigtable: A Distributed Storage System for Structured Data, *Proc. 7th Symp. Operating Systems Design and Implementation*, OSDI '06.

[11] Avinash Lakshman, Avinash Lakshman: Cassandra: A Decentralized Structured Storage System, *ACM SIGOPS Operating Systems Review*, Vol. 44, No. 2, pp. 35–40 (2010).

[12] Philip A. Bernstein and Nathan Goodman: Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, Vol. 13, No. 2 (1981).

[13] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki: Aether: A Scalable Approach to Logging, *Proc. 36th Int. Conf. Very Large Data Bases*, VLDB'10.

[14] Open NAND Flash Interface: OFNI, `http://www.onfi.org/specifications`.

[15] Intel Corporation.: Storage Performance Development Kit, `http://www.spdk.io/`.

[16] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, Yuanzheng Wang: SDF: Software-Defined Flash for Web-Scale Internet Storage Systems, *Proc. 19th Int. Conf. Architectural Support for Programming Languages and Operating Systems* , ASPLOS'14.

[17] Matias Bjørling, Javier González, Philippe Bonnet: LightNVM: The Linux Open-Channel SSD Subsystem, *Proc. 15th USENIX Conf. File and Storage Technologies*, FAST'17.

[18] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, Arvind: Application-Managed Flash, *Proc. 14th USENIX Conf. File and Storage Technologies*, FAST'16.

[19] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, and Bikash Sharma, Moinuddin K. Qureshi: FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs, *Proc. 15th USENIX Conf. File and Storage Technologies*, FAST'17.

[20] Lorenzo Zuolo, Cristian Zambelli, Rino Micheloni, Salvatore Galfano, Marco Indaco, Stefano Di Carlo, Paolo Prinetto, Piero Olivo, Davide Bertozzi: SSDExplorer: a Virtual Platform for Fine-Grained Design Space Exploration of Solid State Drives, *Proc. Conf. Design, Automation & Test in Europe*, DATE'14.

[21] Youngjae Kim, Brendan Tauras, Aayush Gupta, Bhuvan Urgaonkar: FlashSim: A Simulator for NAND Flash-based Solid-State Drive, *Proc. 1st Int. Conf. Advances in System Simulation*, SIMUL'09.

[22] Yohei Hasegawa, Akira Kuroda, Hidenori Matsuzaki, Shigehiro Asano: Architectural Exploration Platform for Solid State Drive Designs, *Proc. 52nd Design Automation Conf.*, DAC'15.

[23] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, Jaehyuk Cha: VSSIM: Virtual Machine based SSD Simulator, *Proc. 29th Int. Conf. Mass Storage Systems and Technology*, MSST'13.