修士論文

SSD を対象とした Key-Value Store のデータ構造に関する研究

指導教官

浅野正一郎 教授

2012 年 2 月 8 日提出

東京大学大学院 情報理工学系研究科 電子情報学専攻 修士課程

学生証番号 48-106436

山田 淳二

目次

第 1 章 序章 6
1.1. はじめに 7
1.2. 本論文の構成 7
第 2 章 NAND Flashメモリの性質 8
2.1. NAND Flashメモリ単体の性質 9
2.1.1. メモリセル 9
2.1.2. メモリチップ 9
2.1.3. DRAMとの比較 11
2.1.4. 書込動作と消去動作 11
2.2. NAND Flashメモリによるストレージ 13
2.2.1. FTL(Flash Translation Layer)の必要性 13
2.2.2. FTLの分類 13
2.2.3. メモリカード向けFTL 13
2.2.4. Page-level FTL 15
2.2.5. SSDのFTL 16
第 3 章 関連研究 18
3.1. Key-Value Storeの研究動向 19
3.2. 主記憶上のKey-Value Store 19
3.3. HDD上のKey-Value Store 19
3.4. NAND Flashメモリ上のKey-Value Store 21
3.4.1. FlashStore 21
3.4.2. SkimpyStash 21
3.4.3. Amazon DynamoDB 21
第 4 章 予備評価 22
4.1. ストレージの性能測定 23
4.1.1. 評価条件 23
4.1.2. 測定結果 23
4.1.3. 考察 23
4.1.4. USBフラッシュメモリの性能測定 30
4.2. 基本的なデータ構造の予備検討 34
4.2.1. ISAM(Indexed Sequential Access Method) 34
4.2.2. B+Tree 34
4.2.3. ハッシュテーブル 38
4.2.4. 基本的なデータ構造のまとめ 40
第 5 章 Bloom Filterを利用したインデックス 41
5.1. Bloom Filter 42
5.2. Bloom Filterの基本的な動作 42
5.3. hBaseでのBloom Filterの利用 43
5.4. 従来のBloom Filterインデックスの課題 43
5.4.1. 主記憶上の探索速度 43
5.4.2. “空振り”による速度低下 45
第 6 章 本研究による提案 48

6.1. 主記憶上インデックス - Bloom Filter Map 49
6.2. “空振り”対策 - 複合ブロック構造 49
第 7 章 提案手法の評価 52
7.1. 主記憶上インデックスの評価 53
7.1.1. 評価条件 53
7.1.2. 評価結果 53
7.2. 複合ブロック構造の評価 59
7.2.1. 評価条件 59
7.2.2. 評価結果 59
7.2.3. 考察 61
第 8 章 システム上での利用形態 66
8.1. システム上での利用形態 67
8.2. 経済的制約 67
8.3. 主記憶のスケーラビリティによる制約 67
8.4. KVSで必要とされる容量比 68
第 9 章 複合ブロック構造の改良 69
9.1. 複合ブロック構造の課題 70
9.2. 改良ハッシュテーブル方式 70
9.3. ページインデックスの集中配置 71
9.4. 先読みによるアクセス効率の改善 71
第 10 章 改良方式の評価 72
10.1. 評価条件 73
10.2. 評価結果 73
10.3. 考察 73
第 11 章 結論 78
11.1. まとめ 79
11.2. 今後の展開 80
謝辞 81
発表文献 82
参考文献 83

図表目次

図 1. NAND Flashメモリのセル構造 9
図 2. NAND Flashメモリのチップレイアウト.....9
図 3. NAND FlashメモリのMemory Array構造 10
図 4. NAND Flashメモリの”ストリング” 10
図 5. NAND Flashのメモリセルサイズ 11
図 6. NAND Flashメモリの書込み操作 12
図 7. NAND Flashメモリの消去操作 12
表 1. NAND Flashメモリのスペックの一例[3] 12
表 2. FTLの種類とアドレス変換表のサイズ 13
図 8. logblockの構造 14
図 9. logblockのswitch操作 14
図 10. logblockのmerge操作 15
図 11. Page-level FTL.....15
図 12. Page-level FTLのGarbage Collection 16
図 13. memcachedのデータ構造 19
図 14. hBaseの概念的データ構造 20
図 15. hBaseの物理的データ構造(hFile) 20
図 16. Hadoop Distributed File System上のhFile 21
表 3. 評価対象SSDのスペック[30][31].....23
表 4. SSDの評価条件 23
図 17. Intel X25-Mの内部構造 24
図 18. SSDのアクセス時間(Writeキャッシュ・イネーブル) 25
図 19. SSDのアクセス時間(Writeキャッシュ・ディセーブル) 26
図 20. SSDアクセス時間のランダム/シーケンシャル比率 27
図 21. SSDの書込み回数対書込み時間(シーケンシャル) 28
図 22. SSDの書込み回数対書込み時間(ランダム) 29
図 23. USBフラッシュメモリのアクセス時間 31
図 24. USBフラッシュメモリの書込み回数対書込み時間(シーケンシャル) 32
図 25. USBフラッシュメモリの書込み回数対書込み時間(ランダム) 33
図 26. ISAM 34
図 27. B+Tree 35
表 5. B+Tree評価条件 35
図 28. B+Treeの特性 36
図 29. B+Treeの特性(拡大) 37
図 30. ハッシュテーブルの構造 38
表 6. ハッシュテーブル評価条件 38
図 31. ハッシュテーブルの充填率対書込み時間 39
表 7. 基本的なデータ構造のまとめ 40
図 32. Bloom Filterの操作 42
図 33. hFile上のBloomFilter.....43
図 34. アドレスの平均間隔 44

図 35. 要素数・ブロックサイズと主記憶インデックスアクセス時間 45
図 36. Bloom Filterインデックスによる”空振り” 46
図 37. “空振り”に伴う平均読出し時間(全データ読出し=100%とする相対値) 47
図 38. Bloom Filter Map 49
図 39. 複合ブロック構造 50
図 40. ブロックの内部構造 51
表 8. 評価環境 53
図 41. 一様分布 55
図 42. Zipfian分布 56
図 43. Latest/Zipfian分布 57
図 44. ブロック数対アクセス時間 58
表 9. 評価条件 59
図 45. ブロックサイズ対アクセス時間 60
図 46. ブロックサイズ対ブロック探索回数 63
図 47. ブロックサイズ対ページ探索回数 64
図 48. ページ探索回数対アクセス時間 65
図 49. SSD容量当たりのDRAM+SSD合計価格 67
表 10. システムあたりの最大接続容量 68
表 11. 平均Key+ValueサイズとSSD/DRAM容量比 68
図 50. 改良ハッシュテーブル方式 70
図 51. ブロックサイズ対アクセス時間 75
図 52. ブロック探索回数対アクセス時間 76
図 53. ページ探索回数対アクセス時間 77

第1章　序章

1.1. はじめに

半導体技術の進展により NAND Flash メモリを利用した大容量の SSD(Solid State Drive)が広く利用されるようになっている. SSD は, 従来の HDD と比べて桁違いの高速アクセス性能を持つことから, データベース分野, 特に, 近年大規模データベースとして利用されることの多い KVS(Key-Value Store)での利用が研究されている.

NAND Flash メモリは, 上書きが不可能で, 消去単位が書込み単位より大きいという特徴を持つことから, 出来るだけデータをシーケンシャルに書込むことが望ましいとされており, Key-Value 対を到着順にシーケンシャルに SSD に書込み, Key と書込み位置の対応を主記憶上で管理する方式が提案されてきた.

本研究では, シーケンシャル書込みと主記憶上インデックスを利用する従来の方式の内, 主に, 主記憶上インデックス構造として Bloom Filter を用いる方式について, 主記憶上のインデックス構造の改良と, 複合ブロック構造によるブロック内探索の高速化によって, アクセス速度を高速化する手法を検討した.

また, Page-level FTL を用いていると考えられる最新の SSD を利用した場合, ページの上書きを行わない限りは, ランダムライト性能の低下は起きないと考えられることから, ランダムライトを前提としたデータ構造の可能性についても検討し, 上述の複合ブロック構造に, 主記憶上インデックスとハッシュテーブルを併用した構造を取入れて, 更に高速で, 主記憶の使用量を減らすことが出来るデータ構造を提案した. この結果, 主記憶上の探索速度では, 従来と比較して 11.8 倍を実現し, Read 時のアクセス速度 65us, 転送レート 250MB/s 程度の安価な一般消費者向け SSD を利用して, 800us 以下の応答速度を実現する Key-Value Store を構成する目途を得た.

1.2. 本論文の構成

本論文の構成は以下のとおりである.

まず, 本章に続く第 2 章では, NAND Flash メモリ単体と, NAND Flash メモリを利用したストレージ装置である SSD(Solid State Drive)が持つ特性について, 既知の一般的事項を述べる. 第 3 章では, 本研究の関連研究について紹介する. 第 4 章では, SSD の特性について測定結果を提示し, 通常のデータベースで使用される各種のデータ構造について, SSD 上で利用する場合の得失について, 評価結果を交えて論じる. 第 5 章では, 先行研究で使用されている Bloom Filter を使用したインデックス方式について検討し, 従来の方式の限界を明らかにする. 第 6 章では, 本研究による提案である主記憶上インデックスと, 複合ブロック構造について詳細に述べる. 続く, 第 7 章では, 本研究による提案について評価結果を提示し, その特性について考察する. 更に, 第 8 章では, DRAM による主記憶及び SSD について, 経済的・技術的特性, Key-Value Store の利用形態から, 主に, DRAM による主記憶と SSD の容量比がどの程度となるかについて論じる. 続く第 9 章では, 第 7 章で明らかになった最初の提案の複合ブロック構造の特性から, その改良方式について述べ, 第 10 章にその評価結果を提示する. 最後に, 第 11 章を本論文のまとめとする.

第2章 NAND Flash メモリの性質

2.1. NAND Flashメモリ単体の性質

本節では，NAND Flashメモリによる SSDを利用したKey-Value Storeを構築するために必要と考えられる，NAND Flashメモリが持つ性質の概容について述べる．本節は，主に，参考文献[1]による．

2.1.1. メモリセル

NAND Flash メモリのセル構造を図 1 に示す．通常の MOS トランジスタのゲート電極の下に，電気的に絶縁された Floating Gate が挟まれた構造となっており，Floating Gate 内の電子の有無で情報を記憶する．絶縁された Floating Gate と外部の電子のやり取りは，トンネル効果などによるもので，高い電圧を必要とする．絶縁された Floating Gate 内の電子の量で MOS トランジスタの閾値電圧が変動するため，Control Gate に適当な電圧を与えて電流を測定すること，Floating Gate 内の電荷の取出しを行うことなく，データの読出しを行うことが出来る．

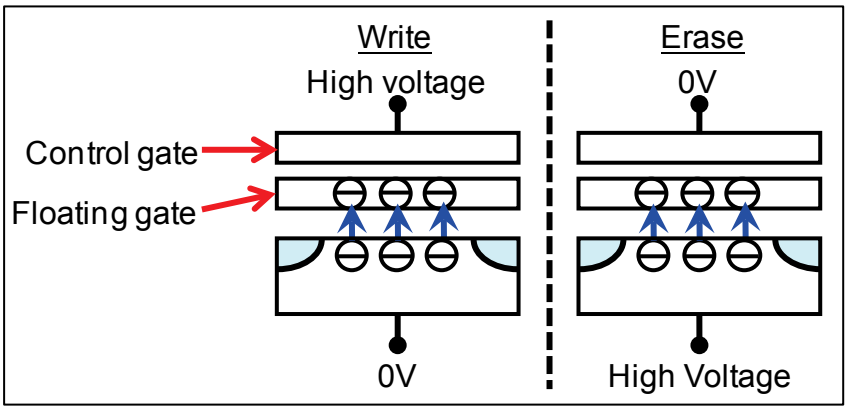


図 1. NAND Flash メモリのセル構造

2.1.2. メモリチップ

図 2 に，NAND Flash メモリの半導体チップの典型的なレイアウトを示す．

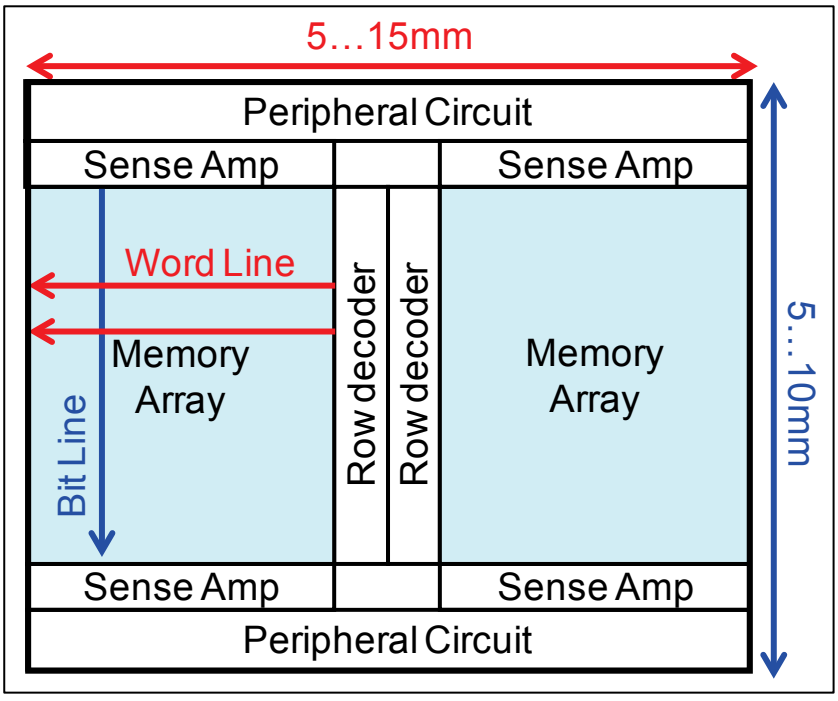


図 2. NAND Flash メモリのチップレイアウト

メモリチップは，Memory Array，ワード線を選択する Row Decoder，ビット線のデータを増幅する Sense Amp，全体の制御・インターフェース機能を持つ Peripheral Circuit 等から構成されている．この内，情報を記憶する Memory Array の構造を図 3 に示す．Memory Array は，ワード線とビット線の交点にメモリセルが形成された構造で，ワード線は，アレイの端から端まで，各メモリセルのゲートに直結されている．

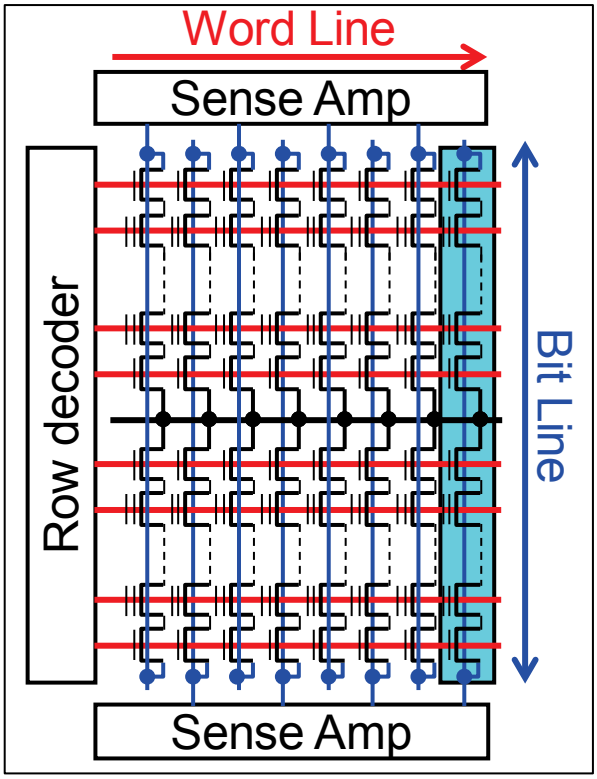


図 3. NAND Flash メモリの Memory Array 構造

ビット線方向には，メモリセルが 16 セル~32 セル，Source/Drain を共有して並んでおりこの単位をストリングと呼ぶ．図 4 に示すように，ストリングは，選択 MOS を介してビット線に接続される．ビットライン-選択 MOS 間のコンタクトは，ストリングごとにあればよいので，コンタクトを最小寸法化する必要はなく，微細化の点で有利である．

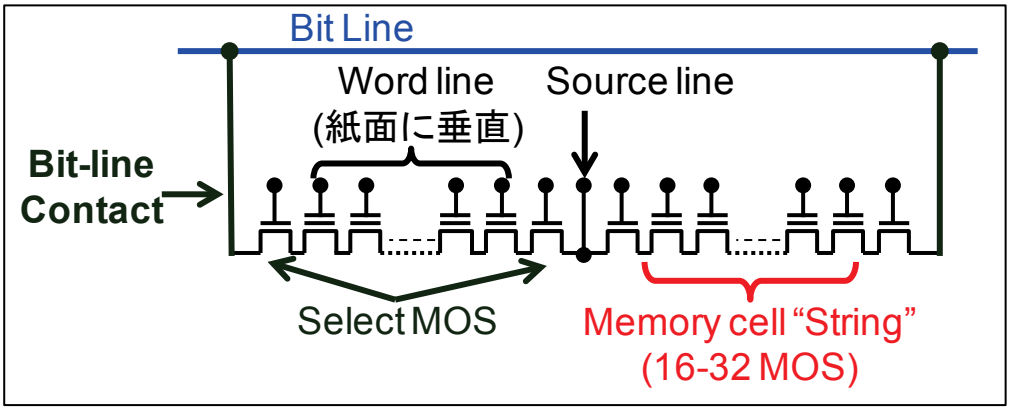


図 4. NAND Flash メモリの”ストリング”

2.1.3. DRAMとの比較

NAND Flashメモリセルは図 5 に示すように，最小加工寸法をFとして，$4F^2$ の面積を持つ．近年のDRAM[2]は $6F^2$ であるので，セル当たりのサイズがDRAMの 2/3 となる．また，NAND Flashメモリでは，1 つのセルへの書込み状態を多値化し，1 セル当たり複数ビットの記録を行うMLC(Multi-Level-Cell)方式が実用となっている．例えば，2 ビット/セルのMLC NAND FlashメモリをDRAMと比較すると，1 ビット当たりの面積は 1/3，3 ビット/セルのMLC NAND Flashメモリでは 2/9 となる．従って，同じ最小加工寸法のプロセス技術・同一面積の場合に，NAND Flashメモリは，DRAMと比較してより大きな容量のメモリを実現することが出来る．

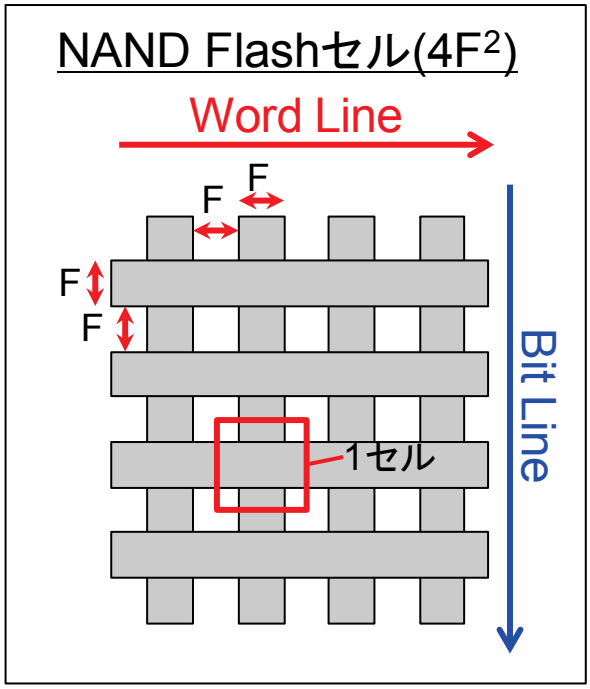


図 5. NAND Flash のメモリセルサイズ

2.1.4. 書込動作と消去動作

NAND Flash メモリに書込みを行う場合のアレイの状態を図 6 に示す．

書込みを行う場合，書込みを行いたいメモリセル群に接続されているワード線に高電圧を加え，同一ストリング上の他のメモリセルには中間電圧を加える．この状態で，各ビット線に高電圧又は 0V を与えると，0V のビット線と高電圧のワード線に挟まれたメモリセルにのみ電子が注入される．従って，書込みの最小単位はワード線の長さで決定されることになる．

次に，NAND Flash メモリを消去する場合のアレイの状態を図 7 に示す．

消去操作では，Floating gate から，Well に電子を引き抜く必要があることから，Well に高電圧が印加する．このため，消去の単位は Well 分離単位となり，必然的に，書込み単位より消去単位が大きくなる．

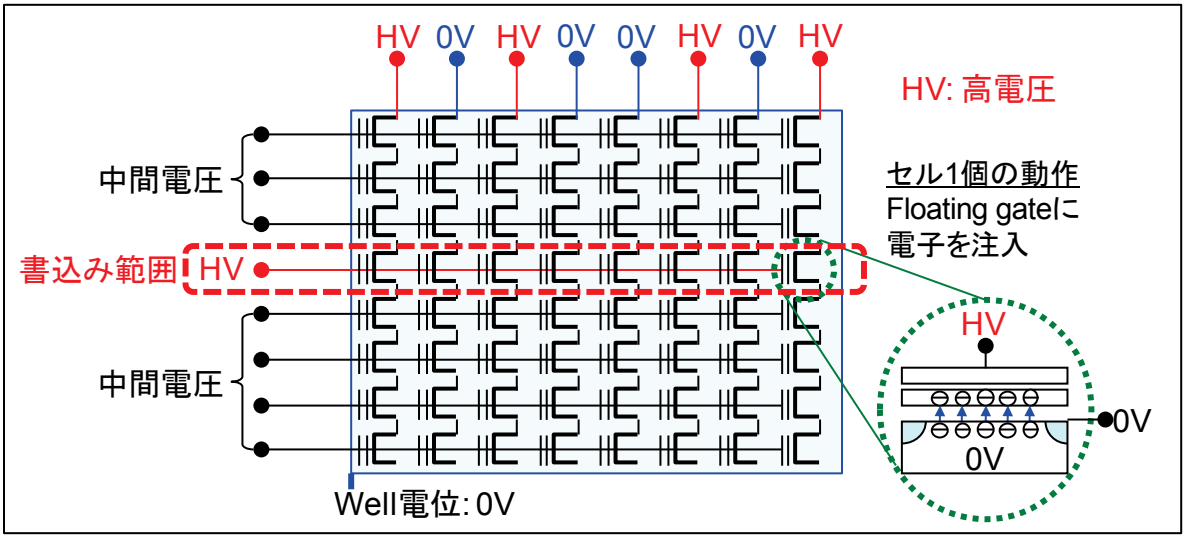


図 6. NAND Flash メモリの書込み操作

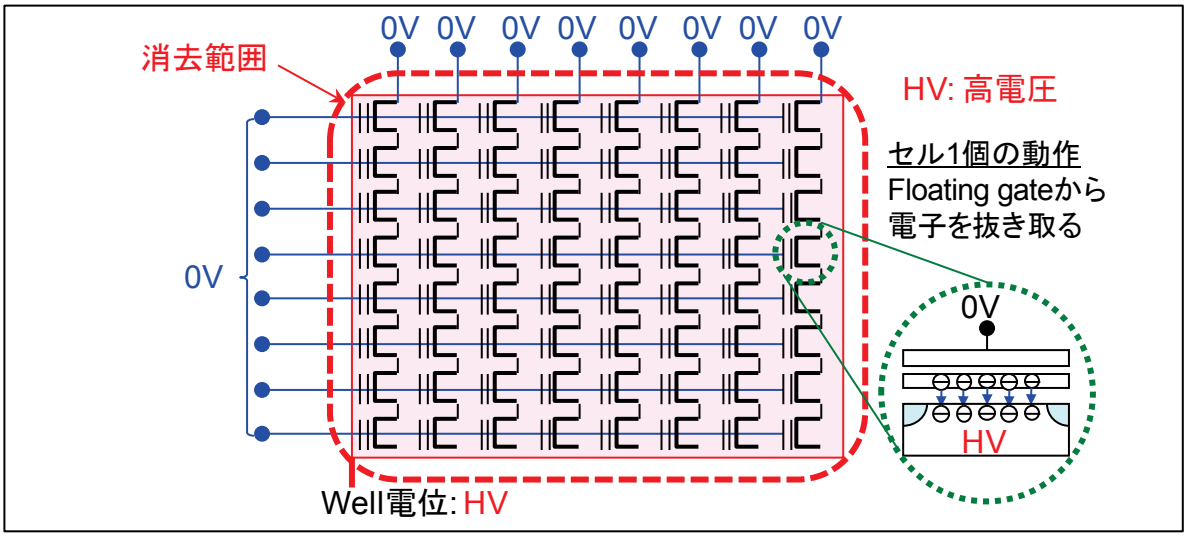


図 7. NAND Flash メモリの消去操作

NAND Flashメモリ単体チップの具体的な製品の一例として，Hynix社製HY27UK08BGFM 32Gビット(4Gx8 ビット)の特性[3]を表 1 に示す．

ここで，ページは読出し及び書込みの最低単位であり，ブロックは消去の最低単位である．表 1 に示すように，消去単位が書込み単位より大きく，消去速度が遅いため，ブロックの一部のページを書換えると，一部のページの書換えのために，ブロック全体の消去と再書込みが必要となり，パフォーマンスが著しく悪化することになる．なお，ページ/ブロックの単位の余分な領域(+64B, +4KB 等)は，ECC パリティ，書込み回数管理，後述の FTL(Flash Translation Layer)のアドレス管理等のために使われる領域である．

表 1. NAND Flash メモリのスペックの一例[3]

Page size	2048B+64B
Block size	128KB+4KB
Total Size	4GB+128KB
Read time(2KB page)	25us
Write time(2KB page)	200us
Erase time(128KB block)	2ms

2.2. NAND Flashメモリによるストレージ

NAND Flash メモリは, システムから直接制御して使われることは少なく, メモリカード, USBフラッシュメモリ, それにSSDといった形態で, 通常は, HDD装置の代替として利用される. 本研究でも, SSD を利用した Key-Value Store を構築することを前提としているので, 本節では, NAND Flash メモリを利用した SSD が持つ特性について検討する.

2.2.1. FTL(Flash Translation Layer)の必要性

先に示したように, NAND Flash メモリの書込み単位(ページ)は 2KB 程度, 消去単位(ブロック)は 128KB 程度と, 通常, 512B を 1 セクタとしていた HDD よりアクセス単位が大きく, 上書き操作が不可能といった特性を持っている. このため, HDD と同一インターフェースで利用するためには, 外部からの, HDD のセクタを単位とした読書きコマンドを, NAND Flash メモリに対する読み・書き・消去コマンドに変換するレイヤが必要となる. また, NAND Flash メモリは, 書込み回数に制限があり, エラーの発生も前提となっているため, 書込み回数管理・書込み回数平準化や, エラー訂正も必要となる. システムと NAND Flash メモリの間で, これらの役割を果たす階層を, Flash Translation Layer(FTL)と呼ぶ.

2.2.2. FTLの分類

FTL には, アドレス変換の方式により, ページを単位としたアドレス変換を行う Page-level FTL と, ブロックを単位としたアドレス変換を行う Block-level FTL の 2 種類がある. 表 2 に, 表 1 に挙げた NAND Flash メモリを利用して, 80GB の SSD を作ることを想定した場合に必要と考えられるアドレス変換表のサイズを計算した結果を示す. 必要な変換表のサイズは, Page-level FTL では 126.6MB, Block-level FTL では 1.5MB と大きな差がある.

表 2. FTL の種類とアドレス変換表のサイズ

FTLの種類	Page-level	Block-level
変換単位	Page	Block
単位サイズ(B)	2,048	131,072
SSD総容量(GB)	80	80
Page/Block数	41,943,040	655,360
アドレスの必要ビット数	25	19
変換表の必要サイズ(MB)	126.6	1.5

2.2.3. メモリカード向けFTL

従来, NAND Flashメモリが利用されるのは, CompactFlash[4], SDメモリーカード[5]といったメモリカードが中心であった. これらのメモリカード上のコントローラーで利用可能なメモリは小規模な内蔵SRAMであるという大きな制約がある. 例えば, Silicon Motion社のSDカード向けコントローラーSM2702 では, 内蔵のSRAMは 100kBに満たない[6]. このような制約から, 従来のメモリカードでは, Block単位FTLを基本とした構造が用いられてきたと考えられる.

一例として, J.Kimらが 2003 年に発表したlogblock[7]の構造を図 8 に示す. logblockでは, NAND Flashメモリの大半をBlock-level FTLで管理し, 数ブロックのみをPage-level FTLで管理する複合構造を取っており, この構造によって, アドレス変換表のサイズを抑えている.

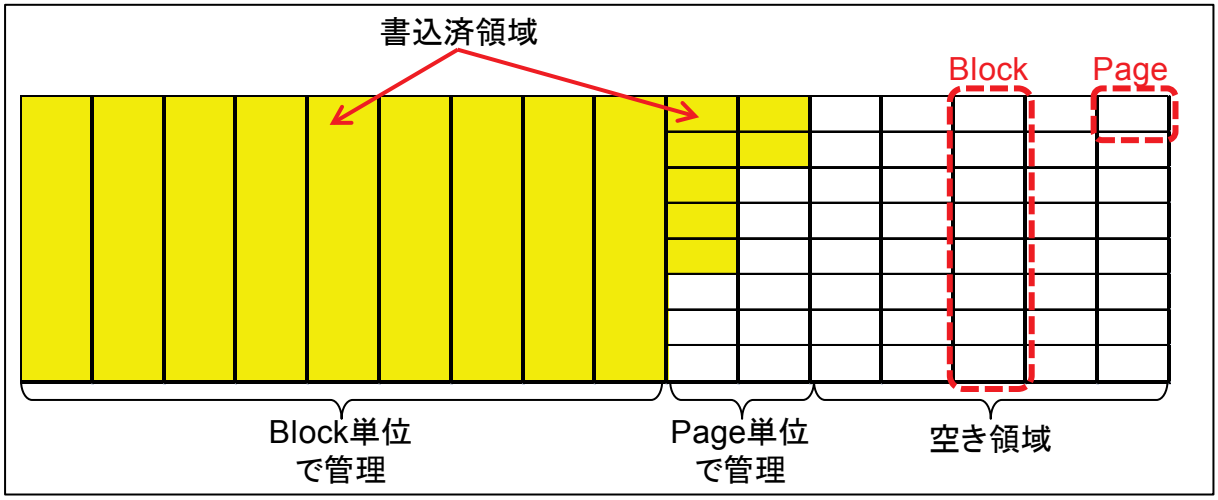


図 8. logblock の構造

logblock では，全ての書込みはページ単位で，Page-level FTL 領域に対して行われ，1 ブロックが埋まったら，Block-level FTL 領域にブロック単位で移動する．ここで”移動”とは，実際には，Block-level FTL でのアドレス変換表の書換えである．

シーケンシャル書込みが行われる場合の logblock の動作を図 9 に示す．ここでは，Block-level FTL 領域のブロックアドレス”5”の各ページに上書きが行われる状態を想定しているが，書込みは全て，Page-level FTL で管理される領域に行われる．Page-level FTL 領域が埋まったタイミングで，Block-level FTL 領域のアドレス”5”と交換する switch 操作を行う必要があるが，シーケンシャルライトが行われている場合は，Page-level FTL 領域は連続するアドレスで順番に埋まっていくから，この交換操作は，単に，Block のアドレス変換表を書き換えるだけで済む．交換前のアドレス”5”のブロックは，消去して空き領域に移されるが，この消去操作も，空き領域が十分残っている限りは，事後的に非同期に行えば問題ない．従って，シーケンシャルライトが行われている限り，FTL に起因して余分な処理が発生することはなく，logblock アルゴリズムは，常に高速な動作を実現することが出来る．

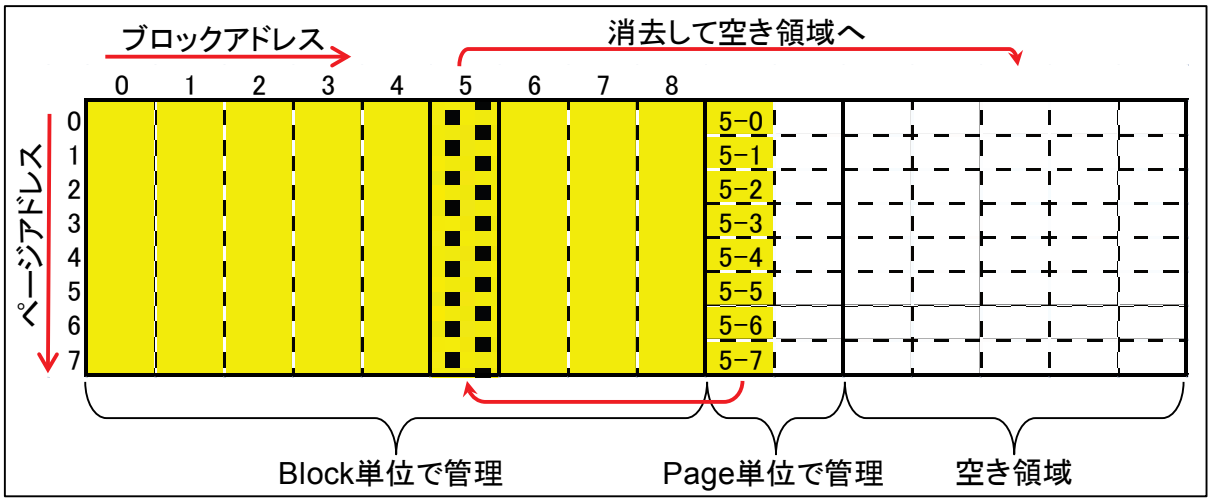


図 9. logblock の switch 操作

これに対して，ランダムライトが行われた場合の動作を図 10 に示す．

この例では，Page-level FTL 領域の一つのブロック内に，ブロックアドレス 5，

7, 2, 6, 3 等のデータが書き込まれている. この状態で Page-level FTL 領域が埋まると, 空き領域のブロックに対して, ブロック 5 のデータを集約したうえで, Block-level FTL 領域と交換する操作が必要となる. これを, logblock では merge 操作と呼んでおり, アドレス変換表の書換えで済んだ switch 操作に比べて, 読出しや書込みが大量に発生して, コストが高い操作となる.

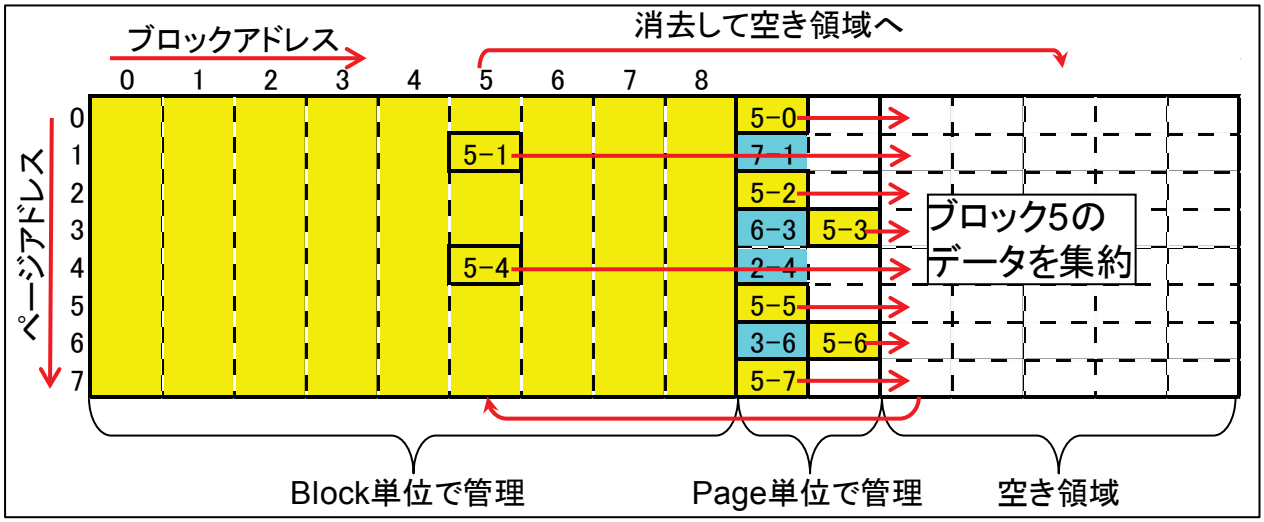


図 10. logblock の merge 操作

以上のように, logblock では, シーケンシャルライトが行われている限りは, FTL に起因する余分な操作は発生せず高速なライトが可能であるが, ランダムライトが行われると, Page-level FTL 領域が埋まるタイミングで, 周期的にコストの高い merge 操作が発生して, 速度が低下するという課題があることがわかる.

2.2.4. Page-level FTL

これに対して, Page-level FTL は, ランダムライトに強い特徴を持っている.

Page-level FTL では, アドレス変換表がページ単位であるため, ランダムに書き込まれたページを, そのままシーケンシャルに書き込んでもアドレス変換表で対応を取ることが出来て, ページを書込む空き領域が存在する限りは, 速度の低下が起こることはない. Page-level FTL で, ランダムライトを行った場合の様子を図 11 に示す. 図の左上から順番にページの書込みが行われており, 各セルの数字は"ブロックアドレス-ページアドレス"を示す.

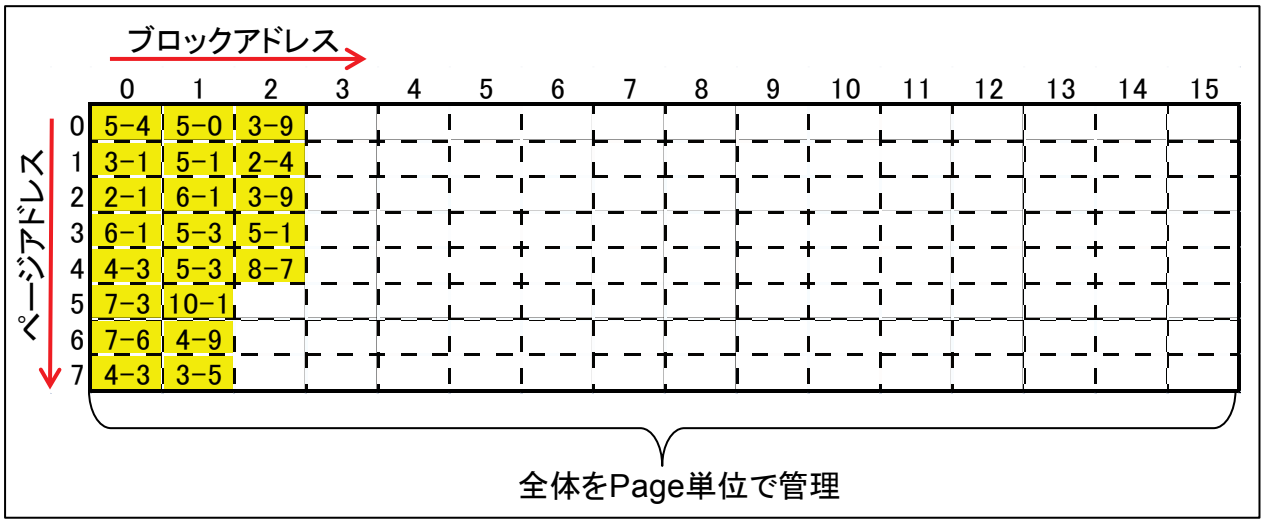


図 11. Page-level FTL

但し，ページに対する上書きは，新しいページへの書込みに変換されるため，上書きが多発する場合には，いずれ，上書き済みデータの消去が必要となる．

ランダムに上書きを含む書込みが行われて，領域が使いきられた状態の例を図 12 に示す．この例では，左上から右下へと，順に，上書きを含むランダムな書き込みが行われている．図 12 中で，”*ブロックアドレス-ページアドレス*”で示すページは，上書きが行われて不要になったページである．

NAND Flash メモリの消去単位はブロックであるため，出来るだけ上書き済みページが多いブロックをいくつか集約，図 12 の例では，例えば，ブロックアドレス 2と3をマージして，空きブロックを作る操作が必要になる．空き領域がゼロになってから，消去やマージ操作を行うと余分な時間が掛かるため，Garbage Collection によって，事前の空きブロックの確保が行われる．

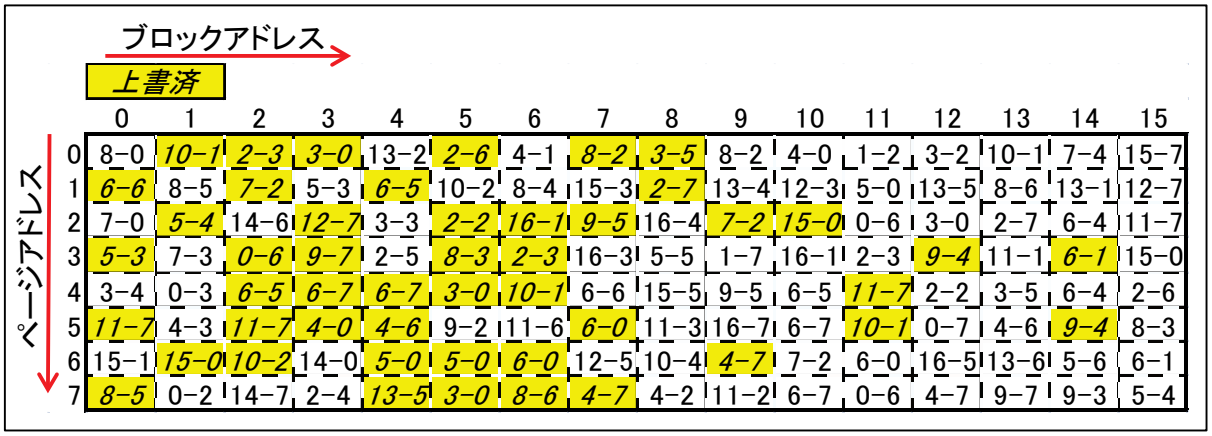


図 12. Page-level FTL の Garbage Collection

図 12 の例では，上書きが行われたページをGarbage Collectionの対象として考えたが，これだけでは十分ではないと考えられている．実際のシステムでは，SSDのようなブロックデバイスは，その上にファイルシステムを構築して利用する場合が多い．通常，ファイルシステムでは，ファイルの削除が行われた場合，インデックス領域のみが操作され，データ領域に対しては何の操作も行われない．この状態では，SSD側では，保持する必要が無い削除済みファイルのデータ部分を保持しようとしてしまうため，Garbage Collectionの効率が低下する．この対策として，未使用のページを，ファイルシステムからSSDに積極的に伝える”TRIM”コマンドが標準化されており，例えばLinuxでは，ext4 ファイルシステムでサポートされている[8]．

2.2.5. SSDのFTL

市販の SSD でどのような FTL が使用されているかは一般に明らかにされていない．また，SSD はいまだ発展途上のデバイスであることから，製品による差も大きいものと考えられる．

Feng Chenらによる市販SSDの調査[9]では，Low-endとされるDRAMバッファを持たないSSDについて，logblockで見られるのと同様の”ランダムライトで周期的に遅くなる”特性が見られる．これは，メモリカードと同様のFTLが用いられている結果であると考えられる．

一方で，同じ調査で，DRAM バッファを持つ Middle-range/High-end の SSD については，ランダムライトによる性能低下が見られない．この理由については明確にはされていないが，外付けの大容量の DRAM バッファが利用可能な SSD では，ランダムライトに強い Page-level FTL を利用している可能性があると

考えられる. 表 2 に示した 80GB の SSD の例では, アドレス変換表の格納に 128MB の RAM バッファが必要となるが, 外付けの大容量 RAM, 例えば 1Gbits DRAM であれば, アドレス変換表全体を RAM 上に格納することが可能である.

また, A.Guptaらが提案するDFTL[10]のように, 有る程度のメモリ容量が確保できれば, アドレス変換表本体はNAND Flash上に保存し, キャッシュしながら利用することも可能である.

A.Gupta らは, DFTL アルゴリズムを, ページサイズ:2KB, 全容量 32GB の SSD に適用した場合の評価を行っている. この場合, 必要となるページアドレス変換表のサイズは,

$$\log_2 \frac{32GB}{2KB} \times \frac{32GB}{2KB} = 49MB$$

であるが, 実際の金融系アプリケーションによるIO Trace[11], SQL Database向けの株式取引をモデルとしたベンチマークであるTPC-H[12]で評価した結果, 金融系アプリケーションによるIO Traceでは 1MB程度, TPC-Hベンチマークでは 4MB程度のRAM容量を確保することで, 十分なページアドレス変換表のヒット率が得られたとしている.

SSD向けのFTLとしては, Block-level FTLを改良したFAST[13], Super Block[14], LAST[15]などの提案もあり, FASTer[16]のように, Page-level FTLより高性能であると主張しているFTLも存在するものの, IBMのX.-Y. Hu.らは, 今後のSSD向けFTLの動向としてPage-level FTLの採用が進み, merge 操作に伴うパフォーマンスオーバーヘッドは減少するものと予測している[17].

第3章　関連研究

3.1. Key-Value Storeの研究動向

本章では, Key-Value Store の関連研究について紹介する.

Key-Value Store は, 従来の SQL 言語で取り扱われるデータベースソフトウェアとは異なり, データの一貫性を保証するような仕組みを持たず, Key に対応する Value を保存するという, 単純なハッシュテーブルと同様の機能に特化することによって, スケーラブルで高性能なシステムを実現している.

3.2. 主記憶上のKey-Value Store

主記憶上に構成されるKey-Value Storeとしてはmemcached[18]が既に実用となっている. memcachedは主記憶の管理に特徴があり, 図 13 に示すように, メモリ全体は 1MB毎のページに分割され, それぞれのページは, 固定サイズの領域に分割して管理している.

memcached に保存されるデータは, そのデータが入る最小サイズの領域を選択して保存される. この方式は, 領域の無駄が一定程度発生するが, 通常の malloc/free 方式と違い, フラグメンテーションが決して発生しないため, ガーベージコレクションに伴う速度低下が起きないというメリットがある.

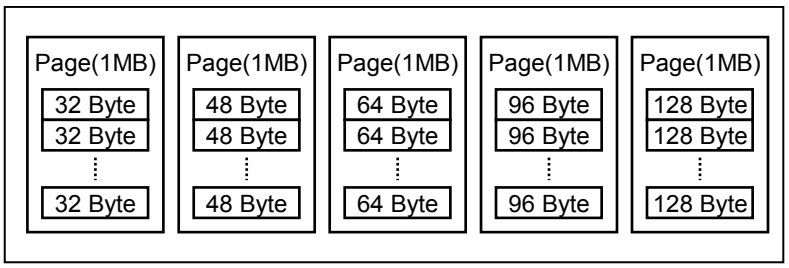


図 13. memcached のデータ構造

memcachedは, 既に, 主記憶型Key-Value Storeとして業界標準の地位を確立しており, memcachedと同一のプロトコルで, Amazon ElastiCache[19]など, クラウド型の商用サービスも行われている.

memcachedは, 主記憶上で動作するため, 当然, 極めて高速であり, 一例として, Facebookでの使用例[20]によると, 35GB/サーバー×800 台の環境で, 20 万アクセス/秒, 平均応答時間 173usを実現しているとされる. 但し, 主記憶のアクセス速度は, 通常 100ns程度[21]であるから, このアクセス速度の大半はシステムやネットワークの遅延などに起因し, 主記憶自体のアクセス速度の寄与はわずかと考えられる. 一例として, Amazon Elastic Cloud2 環境のネットワークレイテンシについて, 145us程度とする測定結果がある[22].

3.3. HDD上のKey-Value Store

HDD 上で構成されるKey-Value Store の代表例としては, Google BigTable[23]が挙げられる. Google BigTableの詳細な実装は開示されていないため, ここでは, そのオープンソース版として開発されているApache hBase[24]について, Apache hBase Book[25]の例を参照して説明する.

Apache は, 概念的には, 図 14 に示すようなデータ構造を持っている. 基本的には, Row Key を Key とする Key-Value Store であるが, Row Key に紐付けされる情報は, Column Family に分割されており, Row Key/Column Family のセットをキーとしてデータにアクセスする. また, 全てのデータには, Time Stamp がつけられており, バージョン管理が行われる. つまり, データの上

書きは行われず追記されるのみである.

Row Key	Time Stamp	Column Family "contents:"	Column Family "anchor:"	
"com.cnn.www"	t9		"anchor:cnnsi.com"	"CNN"
	t8		"anchor:my.look.ca"	"CNN.com"
	t6	"<html>..."		
	t5	"<html>..."		
	t3	"<html>..."		

図 14. hBase の概念的データ構造

hBase では, 図 14 に示したデータを, 図 15 に示すように, Row-Key と Column Family のセット毎に"hFile"として管理する.

hFile 1

Row Key	Time Stamp	Column Family "anchor:"	
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"

hFile 2

Row Key	Time Stamp	Column Family "contents:"
"com.cnn.www"	t6	"<html>..."
	t5	"<html>..."
	t3	"<html>..."

図 15. hBase の物理的データ構造(hFile)

この hFile は, 図 16 に示すような構造でネットワーク上に分散保存される.

hFile は一定サイズを超えると, Row Key/Column Family のセットをキーとして, キーの範囲ごとにネットワーク上の複数の Region Server に保管される. Row Key/Column Family の値と, それを管理する Region Server の対応は, 単一の Name Server が管理している.

クライアントは, Name Server に, アクセスしたい Row Key/Column Family を送り, どの Region Server が管理しているかの情報を得る. クライアントは, この情報をキャッシュするので, Name Server の負荷は問題とはならない. どの範囲のデータがどの Region Server で管理されるかは動的に変わるが, クライアントが誤った Region Server にアクセスした場合は, 単にアクセスが失敗し, 新しい Region Server について, Name Server に問い合わせるだけである.

Region Server は, Hadoop 分散ファイルシステムを利用して, hFile をネットワーク上の複数のサーバーに分散保存する. この時に複数のコピーが取られることによって, データの耐久性を確保し, 同時に, 多数の HDD を並列に使用することで, I/O 性能を確保することが出来る.

データが書き込まれる場合, データは一旦, 主記憶上の MemStore に置かれ, 64MB のデータがたまれば, Hadoop 分散ファイルシステム上に hFile として書き出されていく. 但し, 書込みの安全性を確保するために, 書込み毎に, Write Ahead Log を Hadoop 分散ファイルシステム上に書き出すオプションも持っている. データ参照時には, RegionServer は, まず, MemStore 上にデータがないか探し, MemStore 上にない場合に, Hadoop 分散ファイルシステム上の hFile を一つずつ調べてデータを探索する.

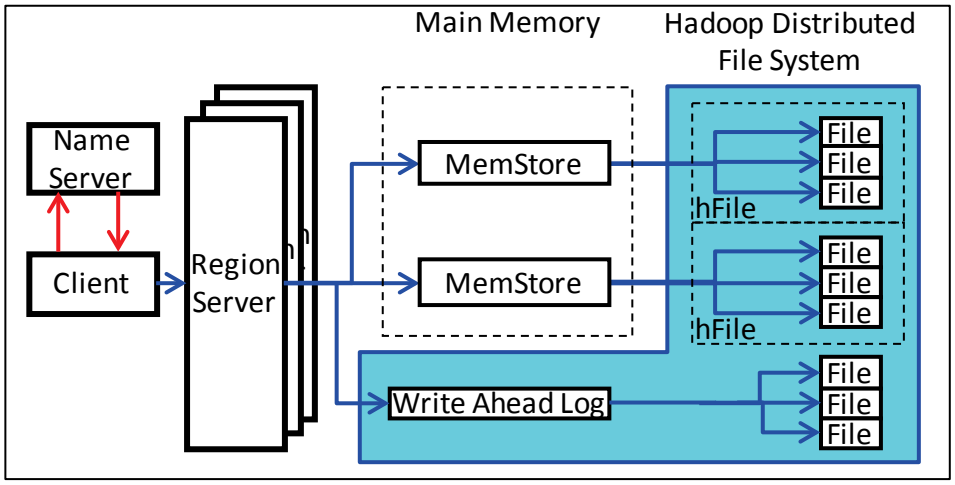


図 16. Hadoop Distributed File System 上の hFile

Brian F. Cooperらによるベンチマーク Yahoo! Cloud Serving Benchmark (YCSB)[26]による評価では，hBaseは，Readアクセス時間が平均 15ms程度，UpdateはMemStore内で完結するため極めて高速であるとしている．

3.4. NAND Flashメモリ上のKey-Value Store

3.4.1. FlashStore

FlashStore[27]は，B.Debnathらが発表したNAND Flashメモリに特化したKey-Value Storeである．到着したデータをNAND Flashメモリ上にシーケンシャルに順次書込みし，キーと記録位置の対応を主記憶上のハッシュテーブルで管理する．FlashStoreでは主記憶の節約のため，キーをハッシュ関数によって 2Bに縮約して記録しており，NAND Flashメモリ上での位置を 4Bのポインタで記録するため，1 要素当たりの主記憶必要量は 6Bであるとされる．これにより，例えば，4GBの主記憶では，715M要素まで管理可能であり，1 要素当たりのサイズを 1KBとすれば，715GBのSSD上のデータが管理可能としている．

3.4.2. SkimpyStash

B.Debnathらは，Flash Storeの改良版として，2011 年 6 月に SkimpyStash[28]を発表している．SkympyStashでは，キーと記録位置の対応を，ハッシュテーブルの代わりに，Bloom Filterによって管理する．B.Debnathらは，1 ブロック当たり 1～32 要素のデータを格納し，1 要素当たり 8bitのBloom Filterを利用することで，約 1B/要素と，高いメモリ効率を実現したとしている．なお，この研究は，筆者が 2011 年 5 月に修士中間発表会で発表したものと同じコンセプトのものであるが，完全に独立して提案されたものであり，以後の研究はこの研究を先行研究として取り扱っている．

3.4.3. Amazon DynamoDB

Amazon DynamoDB[29]は，Amazon社が，2012 年 1 月 18 日に発表した，SSDを使用したKey-Value Storeのクラウド型の商用サービスである．詳細については，まだ十分に明らかになっていないが，1 桁ms程度の非常に高速な応答性能を特徴とし，読出し・書込みスループットに応じた課金を行うシステムであるとされている．

第4章　予備評価

4.1. ストレージの性能測定

本研究を開始するにあたり, NAND Flash メモリを利用したストレージである SSD と, USB フラッシュメモリについて, 性能の予備評価を行った. この予備評価は, 主に, SSD において, Page-level FTL が使用されており, 十分なランダムライト性能が確保されていることを確認することを目的としている.

4.1.1. 評価条件

最新の代表的なSSDとして, Intel社のX25-M(80GB)を選択し, 性能測定を行った. Intel社によるスペック[30][31]を, 表 3 に示す.

実際のシステムでの性能を明らかにするため, 表 4 の条件で, 測定を行った. なお, 先に述べたように, NAND Flash メモリには, 書込みに対して消去が遅いという特性がある. このため, 測定開始時点で, SSD が消去状態にあるか否かで, 速度が変わることが予想される. この問題を解決して安定な測定を行うため, 測定毎に, hdparm を使用し, 消去(TRIM)コマンドによる全消去を行っている.

表 3. 評価対象 SSD のスペック[30][31]

Random Read	35000 IO/sec
Random Write	6600 IO/sec(8GB span) 300 IO/sec(100% span)
Read latency	65us
Write latency	85us
Sequential Read	250MB/s
Sequential Write	70MB/s

表 4. SSD の評価条件

CPU	Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz
Chipset	Intel H67 Express
Mother board	Foxconn H67M-S
HDD I/F	Serial ATA2.0(6Gbps)
Main memory	16GB(DDR3-1333Mbps 64bit Dual channel)
OS	Linux 2.6.35.11-83.fc14.x86_64
fopen option	O_SYNC | O_DIRECT
Write count	2048

4.1.2. 測定結果

測定結果を図 18 及び図 19 に示す. このグラフで, X 軸は, 1 回にアクセスするサイズ(B), Y 軸はアクセス時間(us)を示しており, いずれも log スケールである. 赤が Write, 青が Read であり, それぞれグラフが重なっていてわかりにくいが, マーカーの違いでランダムアクセスとシーケンシャルアクセスを示している. 図 18 と図 19 では, SSD 上のライトキャッシュを切り替えており, ライトキャッシュをイネーブルにした状態が図 18, ディセーブルにした状態が図 19 である.

なお, ライトキャッシュの切り替えは"hdparm –W"コマンドによる. これは, 本来は, システムダウン時のデータ損失を防ぐために, ライトキャッシュを行わないとするオプションである.

4.1.3. 考察

4.1.3.1. シーケンシャル/ランダムアクセスの差

ライトキャッシュイネーブルの図 18 に比べて, ディセーブルの図 19 では, 一回にアクセスするサイズが 4KB～32KB 程度の領域で, ライト性能が悪化している.

しかし，いずれも，シーケンシャル/ランダムライトのグラフはほぼ重なっており差は少ない．この点をより明確にするために，図 20 に，ライトキャッシュイネーブル・ディセーブルそれぞれの条件での，ランダムライト/シーケンシャルライトのアクセス時間の比率を示すが，1 回にアクセスするサイズで多少違うものの，いずれも，ランダム・シーケンシャルライトによる性能の差はほとんど認められない．

SSD のランダムライト性能向上には，2.2.4 で述べた Page-level FTL 以外に，ライトキャッシュによる方式も考えられるが，この結果からは，ライトキャッシュによって，ランダムライトの性能が向上している訳ではないことがわかる．

また，Page-level FTL を使用する際のアドレス変換表は，DRAM バッファの上に配置されるが，この変換表は電源喪失で失われても，SSD のページの予備領域に記録されたアドレスから復元することが出来る．従って，ライトキャッシュディセーブルを指示された場合に，Page-level FTL のアルゴリズムを変更する必要はないと考えられる．これらのことから，intel 社は明確にしていないものの，本製品のランダムライト性能は，Page-level FTL によって実現されている可能性が高いと考えられる．

4.1.3.2. アクセス時間のサイズ依存性

アクセス時間が一回にアクセスするサイズに反比例すると考えた場合，図 18 及び図 19 のグラフは直線になるが，1 回にアクセスするサイズの小さい領域で，直線から外れている．この特性は，以下のように説明をすることができる．まず，4096B～16384B のアクセス時間は一定に近いように見える．Intel X25-M は，図 17 に示すように，内部的に NAND Flash に対して，10 チャンネルの同時アクセスを行っているとされ，同時アクセスするチャネル数がこの範囲で可変であるとすれば説明がつく．2048B 以下の領域では，ライト速度は 4096B よりもさらに遅くなっており，Read-Modify-Write を行っていることが推測される．

図 21 及び図 22 に示すのは，書込み 1 回毎の書込み時間をグラフにしたものである．図 21 がシーケンシャルライト，図 22 がランダムライトの結果であり，各系列が，1 度の書込み量(512B~16384B)を示している．一度の書込み単位が 2048B 以下の領域では，周期的に書込みが遅くなる特性を示している．これは，ページサイズ以下の書込みとなるために，定期的にデータをページ単位にマージするなどの動作が発生しているものと考えられる．これに対して，4096B 以上では，シーケンシャルライト，ランダムライトの差が全く見られず，logblock 等で特徴的な周期的にライトが遅く特性は見られない．この点からも，本製品が Page-level FTL を採用していることが推定される．

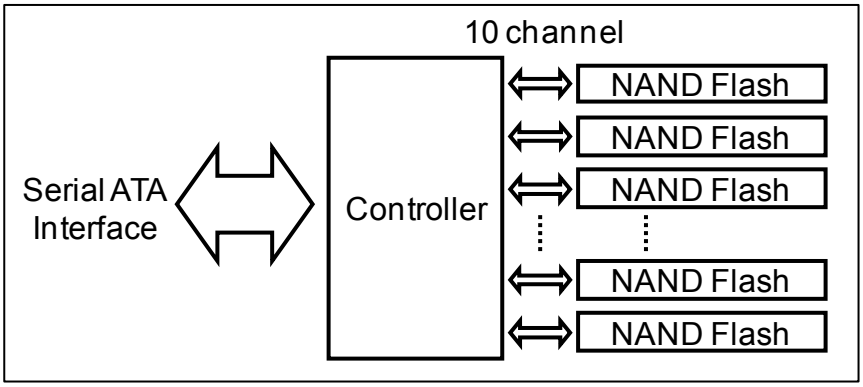


図 17. Intel X25-M の内部構造

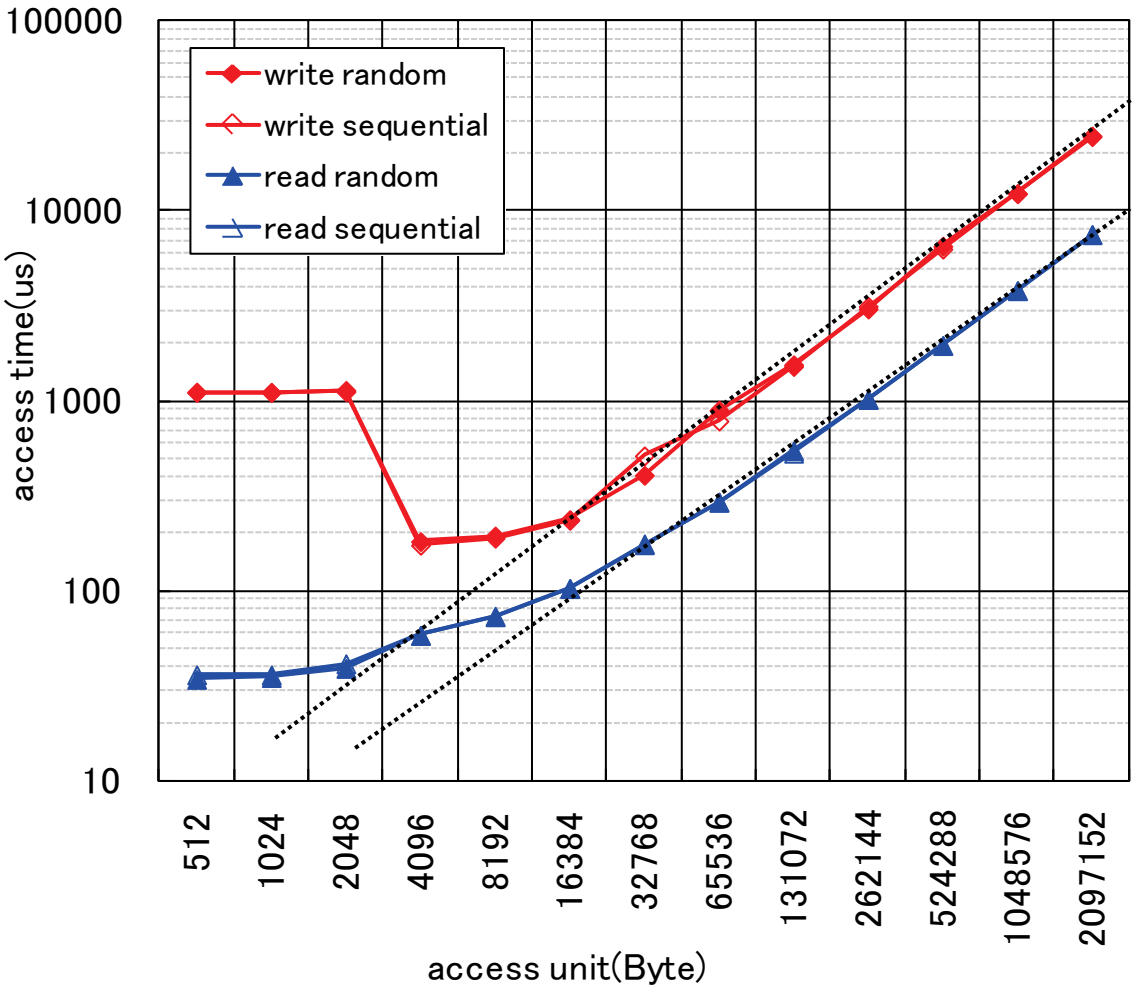


図 18. SSD のアクセス時間(Write キャッシュ・イネーブル)

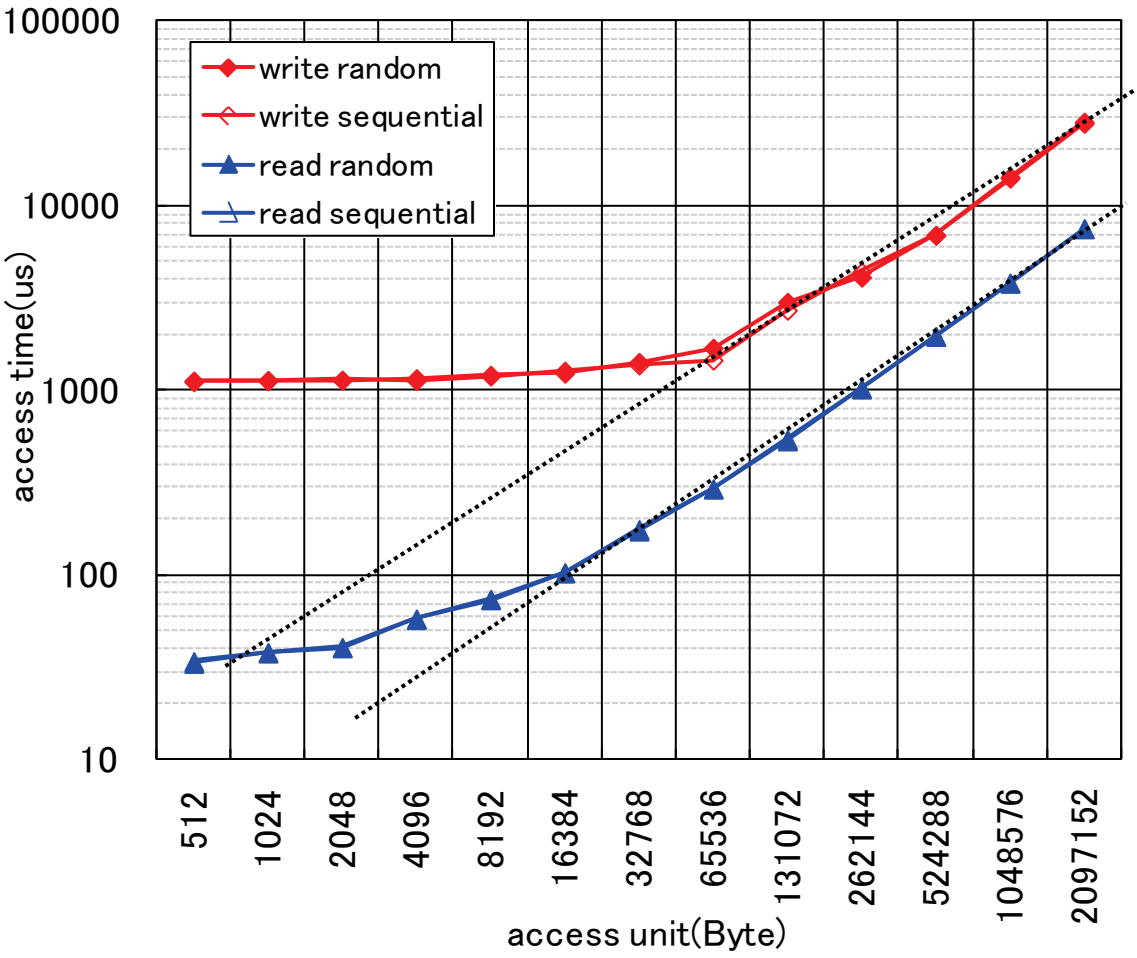


図 19. SSD のアクセス時間(Write キャッシュ・ディセーブル)

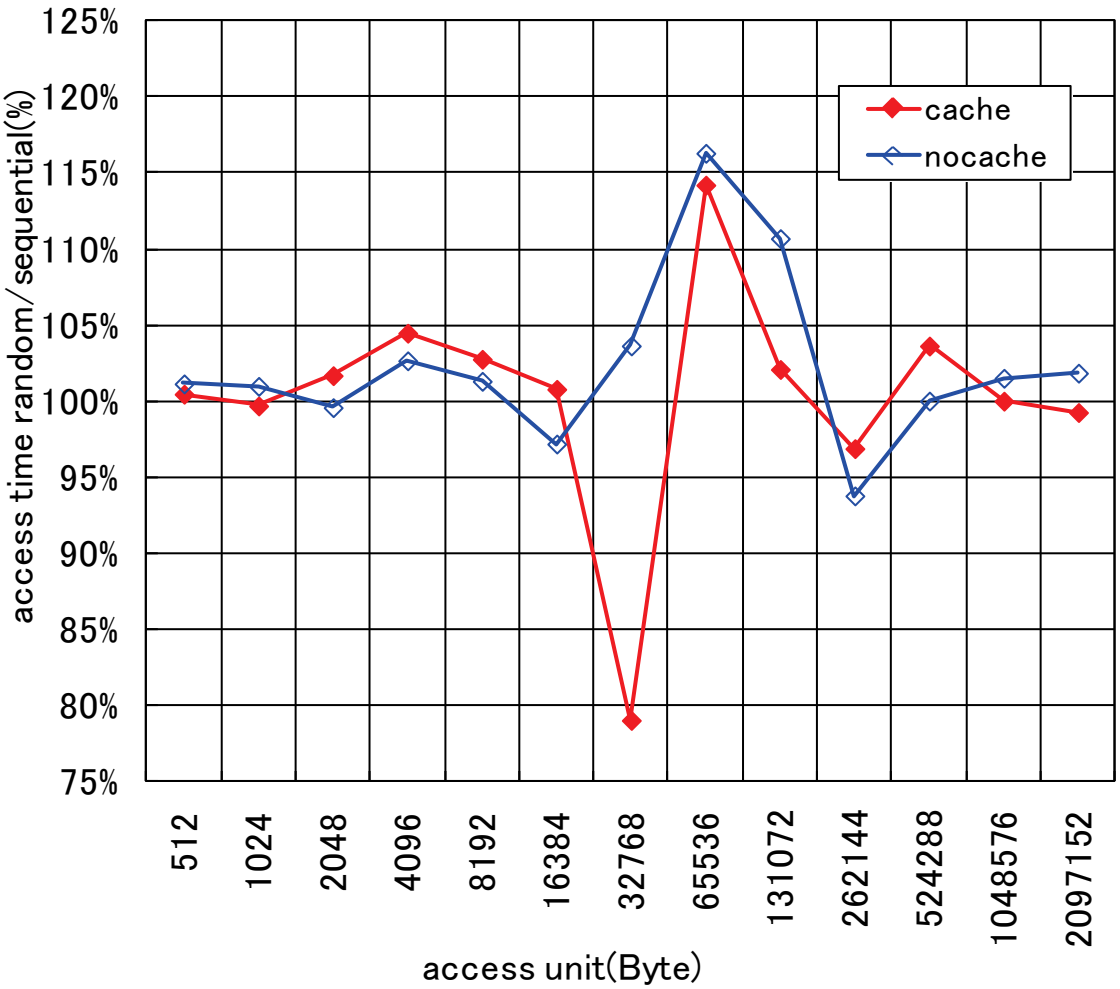


図 20. SSD アクセス時間のランダム/シーケンシャル比率

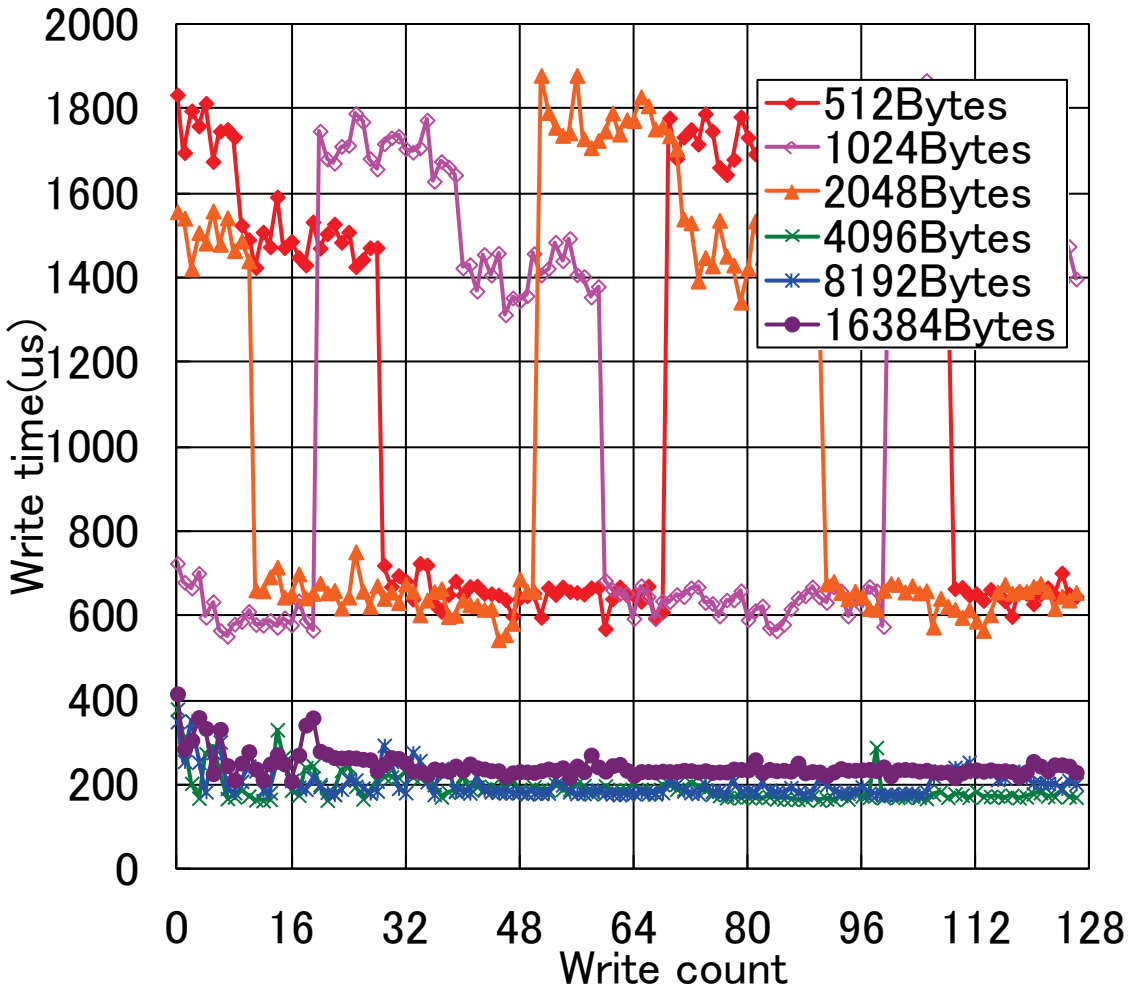


図 21. SSD の書込み回数対書込み時間(シーケンシャル)

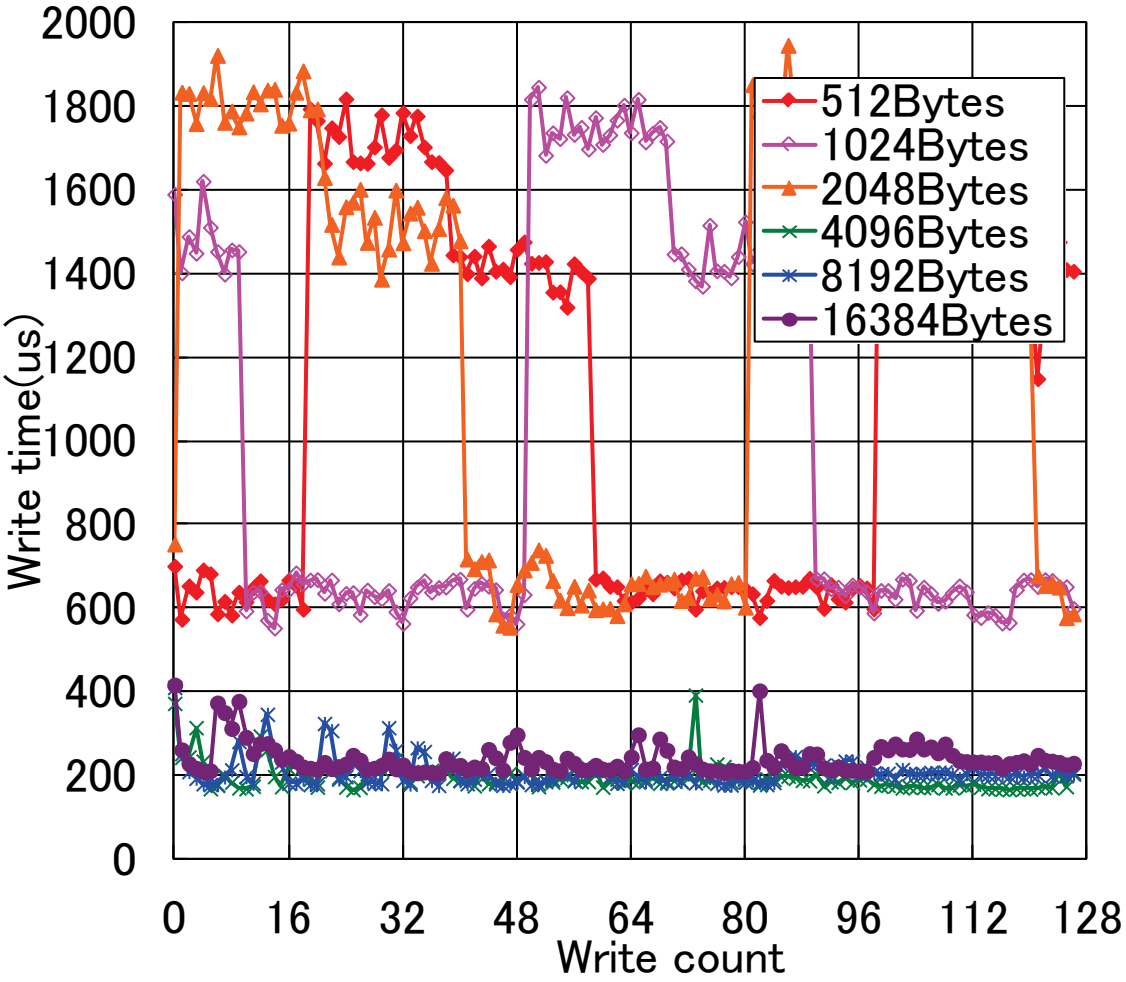


図 22. SSD の書込み回数対書込み時間（ランダム）

4.1.4. USBフラッシュメモリの性能測定

いわゆる USB フラッシュメモリについても，比較的高速な USB3.0 のメモリを使用して，同じ環境で性能の測定を行った．結果を図 23，図 24，図 25 に示す．

図 23 に示すのは，1 回にアクセスするサイズとアクセス時間の対応である．これを，SSD の結果である図 18 及び図 19 と比較すると，SSD とは異なり，シーケンシャルアクセスとランダムアクセスの性能が大きくかい離している．特にライトでは，ランダムアクセスで 2 桁近い性能低下が見られる．

また，書込み 1 回毎の書込み時間をグラフにした図 24，図 25 を見ると，シーケンシャルライトである図 24 では，1 回にライトするサイズによっては周期的なライト時間の変動が見られないが，ランダムライトである図 25 では，サイズによらず，周期的なライト時間の変動が見られる．

USB フラッシュメモリは，SSD とは異なり，大容量の DRAM バッファを使用していないため，Page-level FTL を使用することは困難であると考えられ，これらの特性は，前述の logblock のような，Block-level FTL の影響である可能性が高いと判断できる．

なお，USB フラッシュメモリに関しては，SSD とは異なり，消去(TRIM)コマンドを受け付けないため，代わりに dd コマンドによるゼロ書込みを行っている．この点で，測定が厳密さを欠いている点に留意する必要がある．

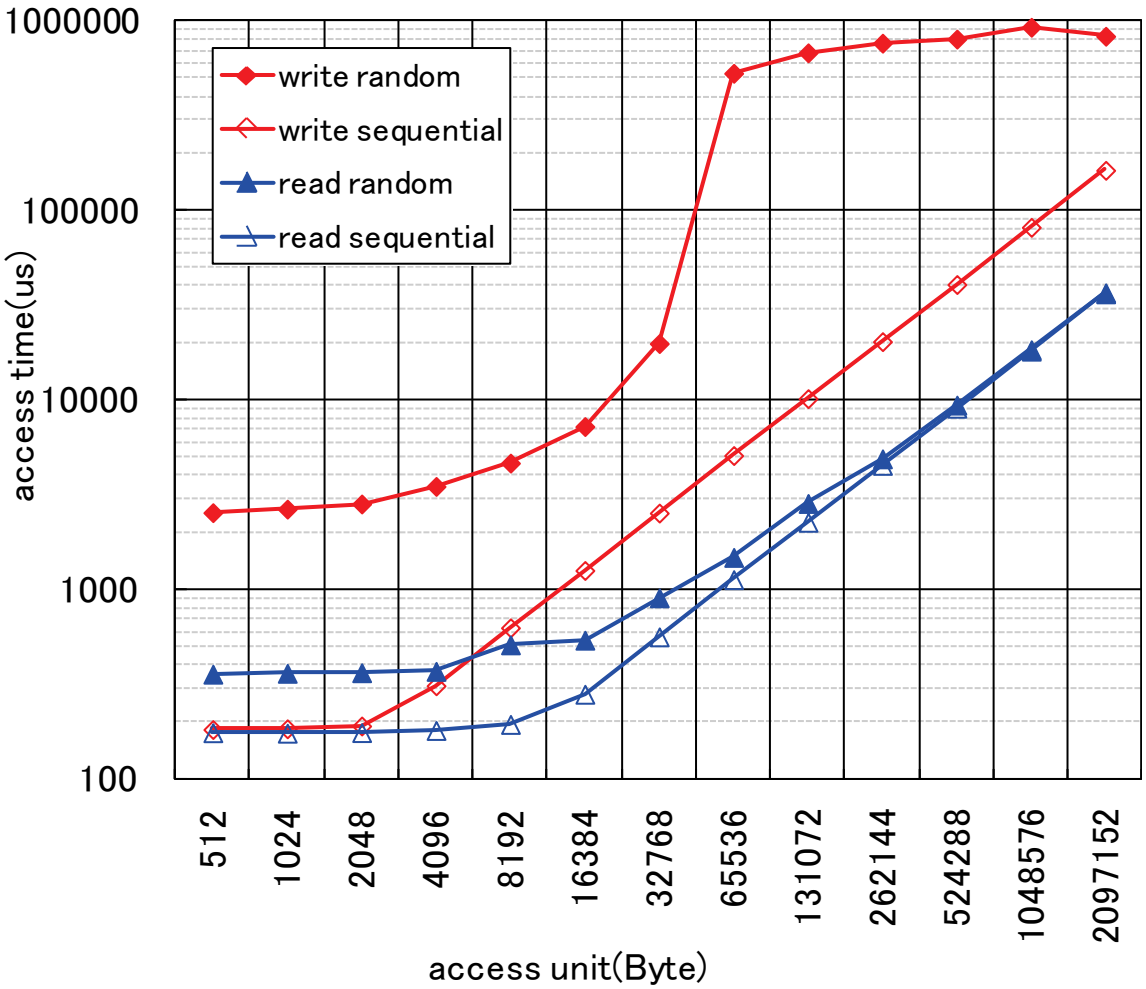


図 23. USB フラッシュメモリのアクセス時間

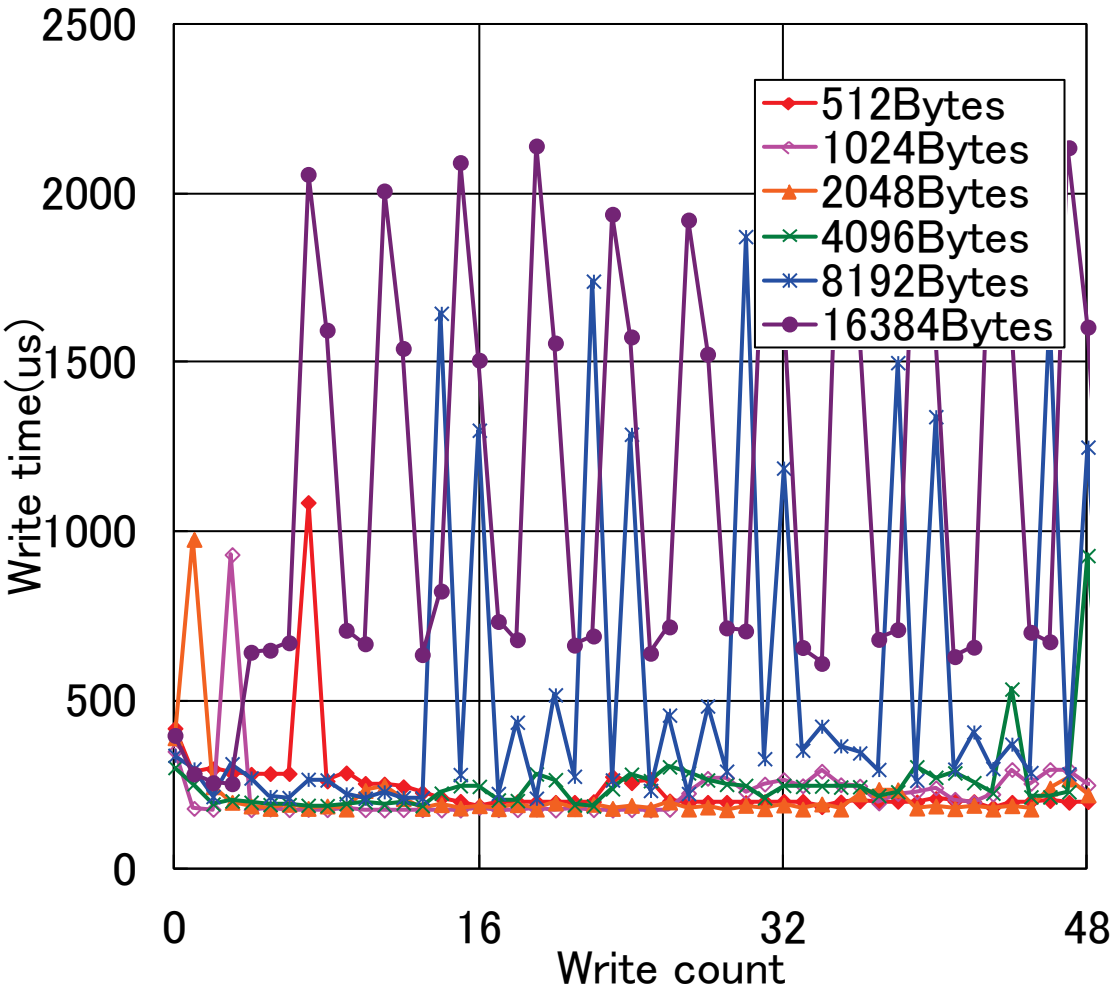


図 24. USB フラッシュメモリの書込み回数対書込み時間（シーケンシャル）

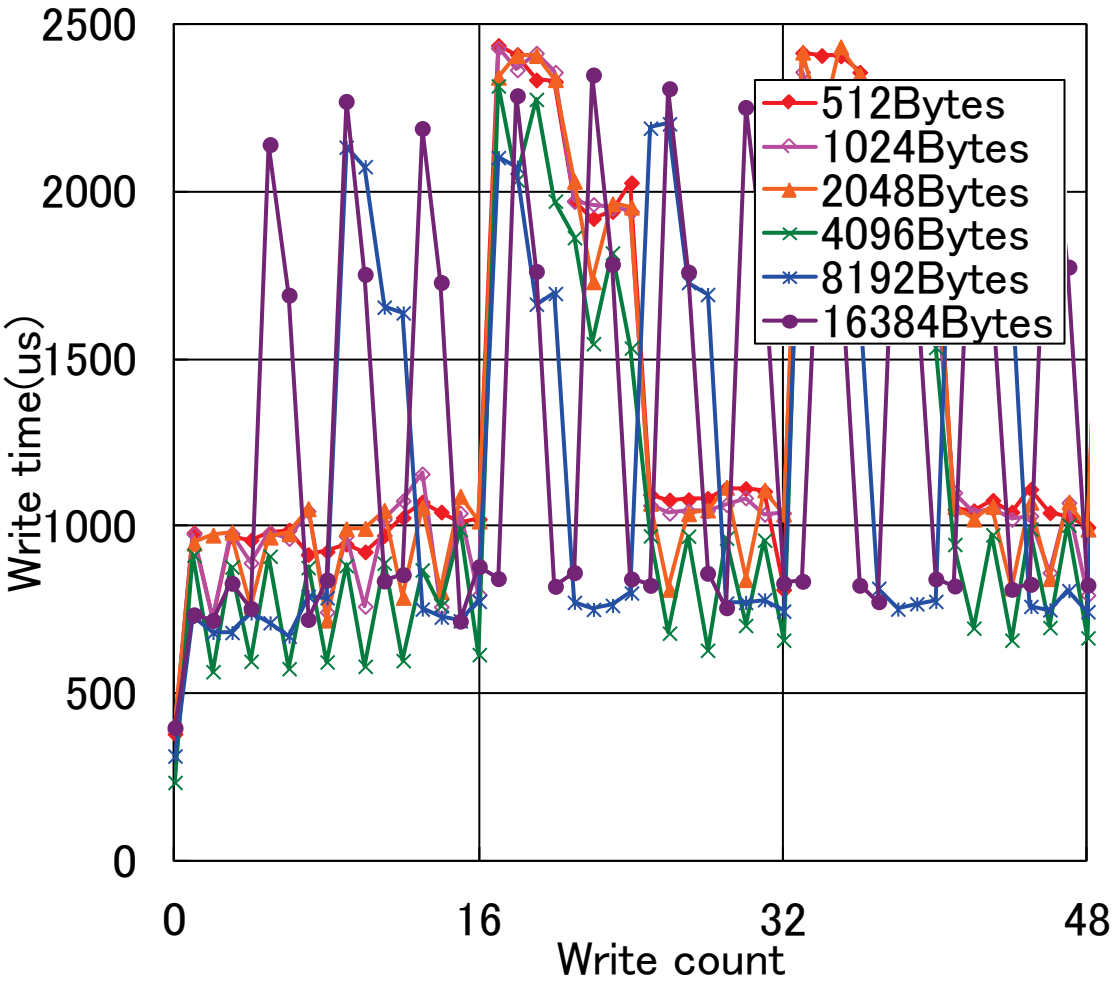


図 25. USB フラッシュメモリの書込み回数対書込み時間（ランダム）

4.2. 基本的なデータ構造の予備検討

次に, SSD 上に Key-Value Store を実装するため, 従来, データベース分野で利用されてきた ISAM, ハッシュテーブル, B+Tree の 3 種類の基本的なデータ構造について予備検討を行った.

4.2.1. ISAM(Indexed Sequential Access Method)

ISAM 方式は, 先行研究の FlashStore[27], SkimpyStash[28]で, SSD 上の Key-Value Store のデータ構造として用いられている方式である.

4.2.1.1. データ構造

ISAM は, 図 26 に示すデータ構造であり, Key-Value 対を, 到着順にシーケンシャルに SSD に書き込んでいく方式である. これと同時に, Key と, 書込み位置の関係を主記憶上にインデックスとして格納する.

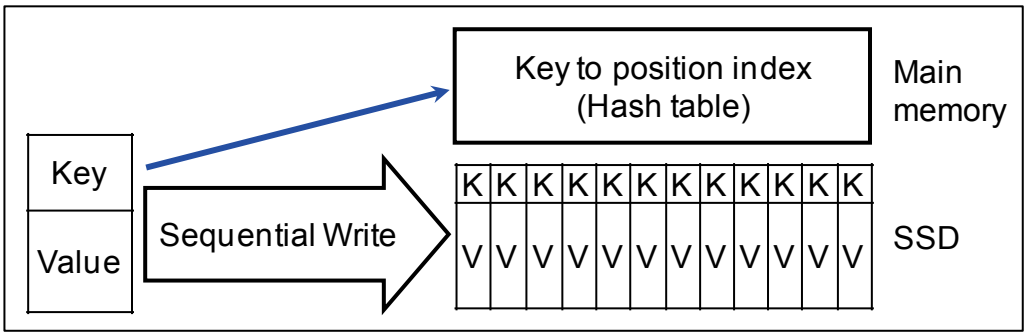


図 26. ISAM

書込みは, シーケンシャルアクセスであるため, ランダムライトに弱いという SSD の欠点は完全に回避される. 読出しも, 主記憶上のインデックスで位置を高速に特定できるため, パフォーマンスを出しやすい.

欠点としては, 主記憶上のインデックスが巨大化することである. このインデックスは, 例えばハッシュテーブルで構成されるが, 最低でも, 各要素の”Key＋アドレス”を記録する必要があり, Key 長を 8 バイト, アドレス 4 バイトとすると, 1 要素当たり最低 12 バイトが必要となる. これは, 1 要素当たりのデータサイズが 1000 バイトとすると, 記録位置を管理するハッシュテーブルに, 最低でもデータ容量の 1.2%が必要であることを意味する. 当然, 1 要素当たりのデータサイズが減るほど, Key 長が長くなるほど, 必要な容量は更に増加する.

4.2.1.2. 評価

ISAM の書込み動作は単なるシーケンシャルライト, 読出し動作は単なるランダムリードであり, SSD の性能測定結果から容易に性能を予測することが可能であるため, 評価は省略する.

4.2.2. B+Tree

4.2.2.1. データ構造

B+Tree は, B-Tree 系の木構造の一種であり, 図 27 に示すように, データは葉ノードのみに記録され, 根から葉の手前までのノードには, キーとポインタが記録される構造である. B+Tree は, HDD 上の関係データベースで広く利用されるデータ構造であるが, 書換え操作に向かない Flash メモリ上で使用する場合の欠点として, データページの変更に対して, インデックスページの書換えが発生することが挙げられる.

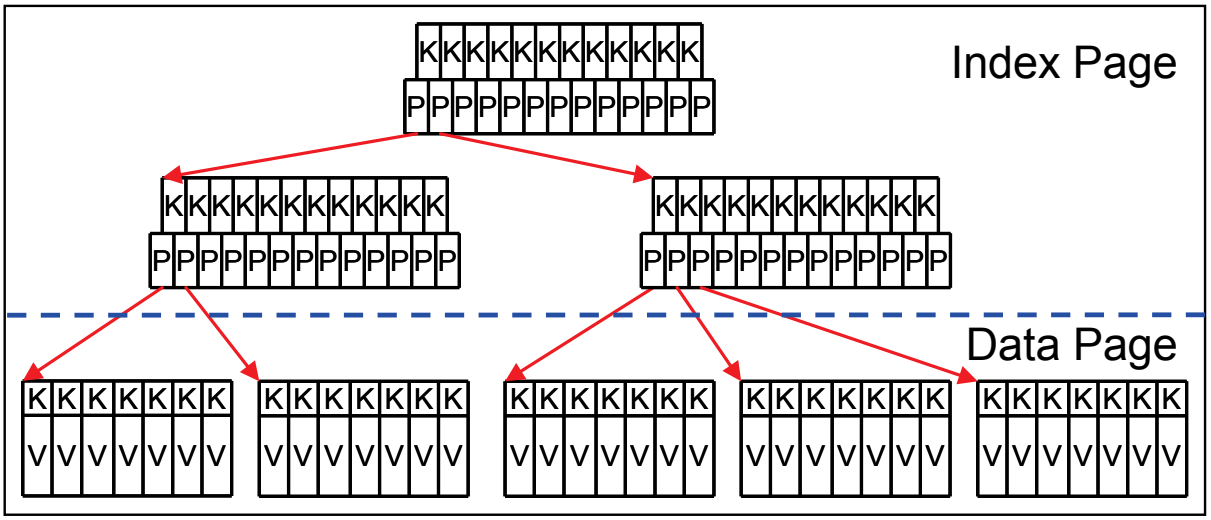


図 27. B+Tree

4.2.2.2. 評価

B+Tree について，表 5 に示す条件で簡単な評価を行った．

表 5. B+Tree 評価条件

Java環境	Open JDK IcedTea6 1.9.7 java version 1.6.0_20
書込み方法	RandomAccessFile class “rw” option “sync” every write
書込み回数	8192回
Key/Valueサイズ	400Bytes

評価結果を図 28 及び図 29 に示す．X 軸はライトの回数，Y 軸に Write 時間(ms)であり，図 28 は全体，図 29 は図 28 の一部の拡大である．図 29 の拡大されたグラフを見て分かる通り，大体の場合の 20ms 程度でライト出来ているものが，一定の間隔で 600ms など極端に遅くなる．極端に遅くなる場合のライト時間は，図 28 の全体のグラフから分かるように徐々に遅くなった後，2650 回，5200 回付近でリセットされるような特性となっている．

これは，インデックスページに対する上書き操作が大量に発生し，SSD の FTL が，Garbage Collection 等を行っているためと考えられる．

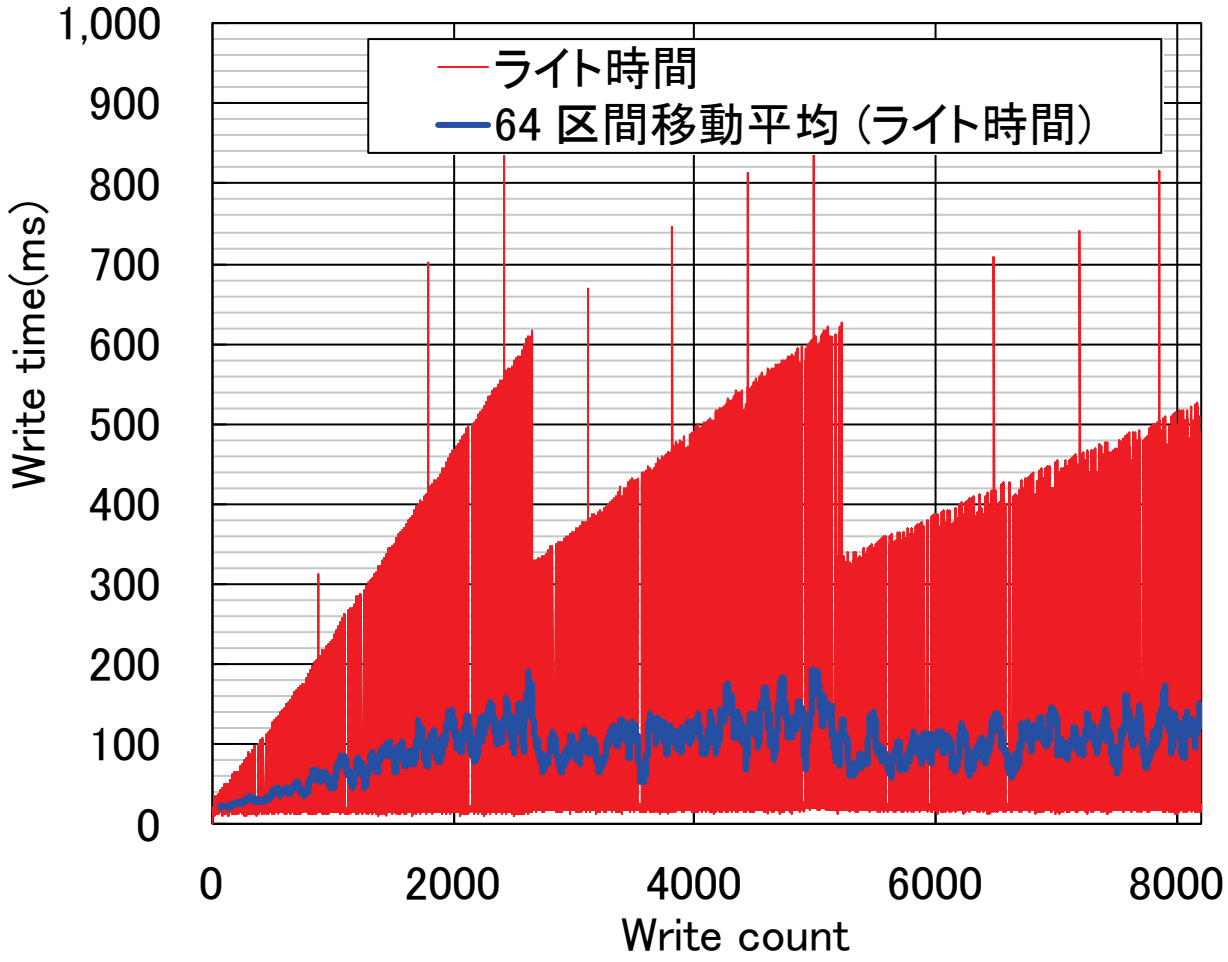


図 28. B+Tree の特性

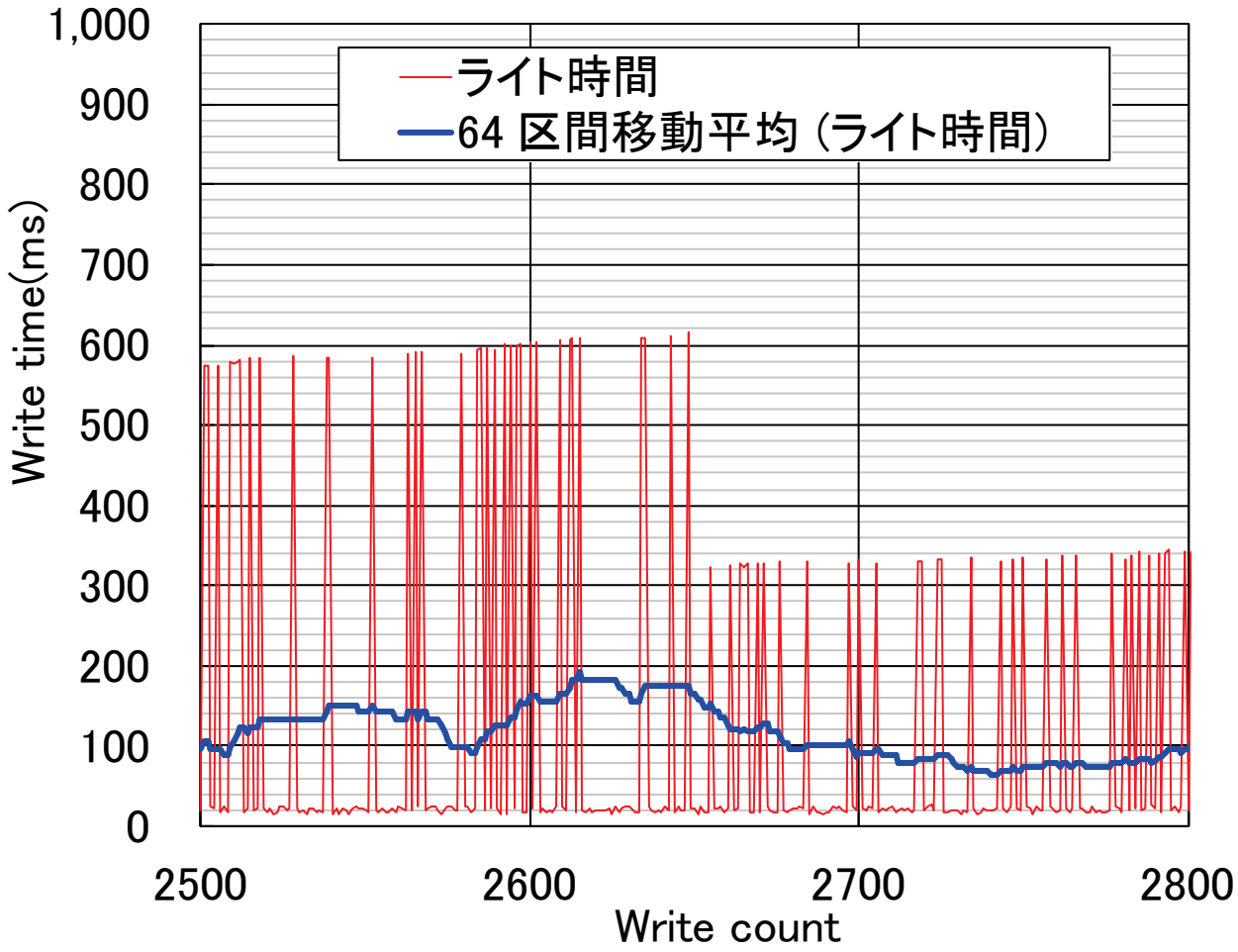


図 29. B+Tree の特性(拡大)

4.2.3. ハッシュテーブル

ハッシュテーブルは，図 30 に示すデータ構造であり，Key-Value 対を，Key から求めたハッシュ値が示す場所に格納する．ハッシュ値の衝突が発生した場合が問題となるが，今回の評価では，空いている領域を単純にリニアにスキャンする方式をとった．

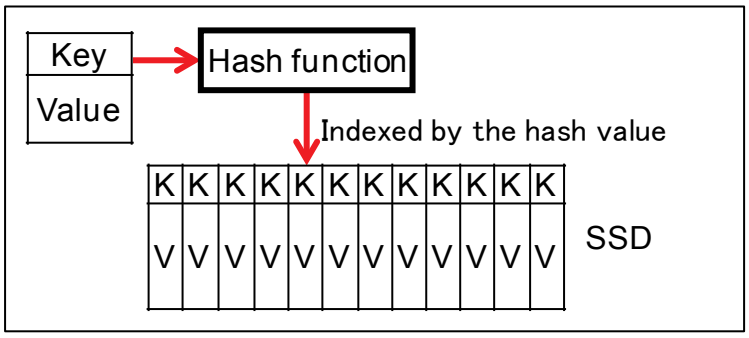


図 30．ハッシュテーブルの構造

ハッシュテーブルの利点としては，主記憶上に，記録位置を保存する必要がないことが挙げられ，主記憶の量によって，管理できるデータの量が制限されない．欠点としては，衝突時の処理がパフォーマンス低下を招くこと，予めハッシュ関数の出力であるアドレスが取りうる範囲を決めなくてはいけないことから，スケーラブルでないといったことがある．また，書込みがランダムになり，ランダム書込み性能が要求されることも欠点だが，ランダム書込み性能に優れた最近の SSD ではこの点は問題にならない可能性もある．

4.2.3.1. ハッシュテーブルの評価

ハッシュテーブルは主記憶の量が少なくて済むメリットがある一方，ランダムアクセス性能の問題が考えられたため，実際に，表 6 に示す条件で実験を行った．

表 6．ハッシュテーブル評価条件

Java環境	Open JDK IcedTea6 1.9.7 java version 1.6.0_20
書込み方法	RandomAccessFile class “rw” option “sync” every write
書込みサイズ	8GB
Key/Valueサイズ	16B(Header)+8B(Key)+1000B(Value)

結果を，図 31 に示す．

図 31 は，書込み回数に対するアクセス時間の変化であり，X 軸は，最大書込み数に対する書込み数の比率，つまり，使用領域(8GB)に対する充填率を単位としている．Y 軸は書込み時間(us)であり，10000 書込み毎の平均値を示している．図 31 から明らかなように，書込み量が増加するに従って，アクセス時間が加速度的に増加している．この評価では，アクセス時間の過去 16 回平均が 50ms を超えた時点で測定を停止しているが，約 74%の時点で書込みが停止している．しかし，これは，ハッシュの衝突に起因するものと考えられ，SSD 自体のランダムライト性能の低下は認められない．2.2.4 で明らかにしたように，Page-level FTL を使用している SSD では，ページの上書きが発生しない限り，ランダムライト性能の低下は発生しないと考えられ，ページの上書きが，ISAM 方式に比べて余分に発生しないという前提があれば，基本的なデータ構造としてハッシュテーブルを使用することも可能であると考えられる．

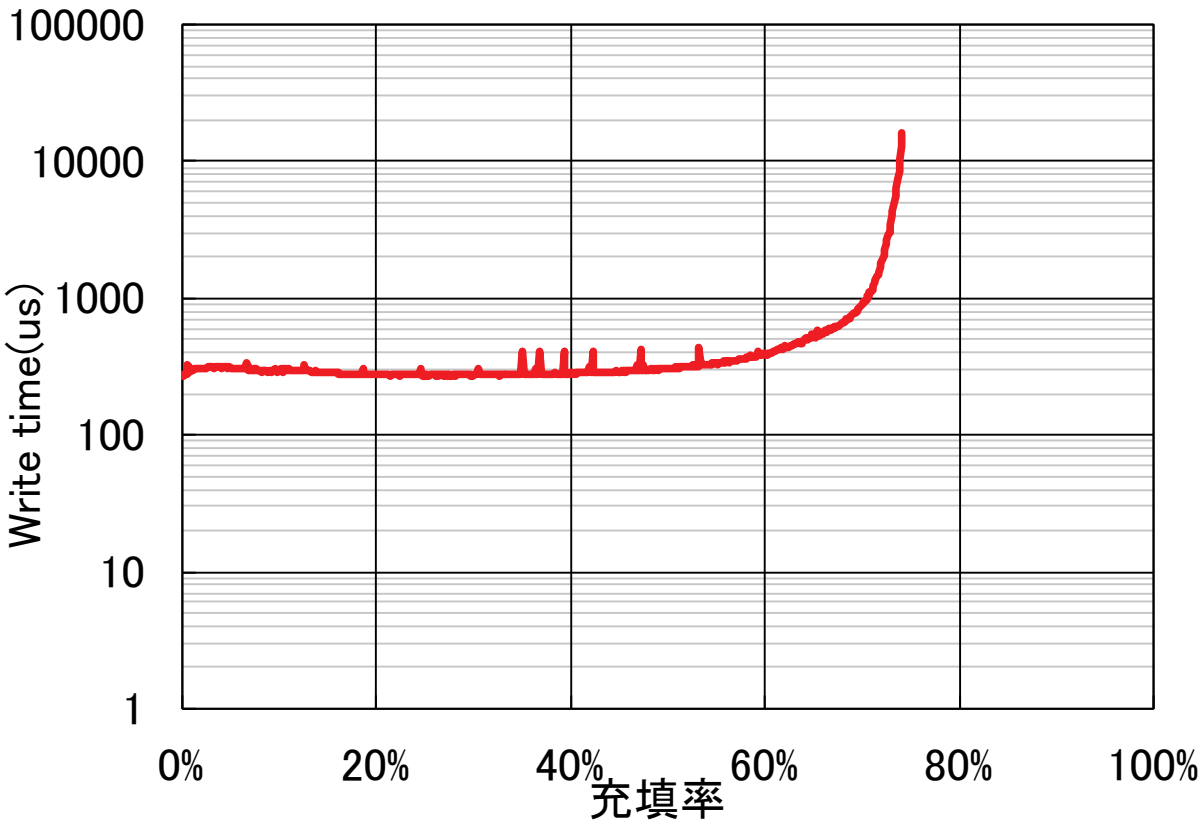


図 31. ハッシュテーブルの充填率対書込み時間

4.2.4. 基本的なデータ構造のまとめ

基本的なデータ構造の特徴，利点・欠点を表 7 にまとめる．出来るだけ多くの SSD 上のデータを，少ない主記憶で管理するという目標に対しては，主記憶上のインデックスが巨大となる ISAM は適さず，インデックスが必要ないハッシュテーブルが最も適している．

表 7. 基本的なデータ構造のまとめ

データ構造	Key/Value の保存位置	インデックス	利点	欠点
ISAM	到着順	主記憶上	シーケンシャルライト	主記憶上インデックスが必要
Hash Table	Key の ハッシュ値	無し	インデックス不要	ランダムライトが必要 スケーラビリティ無し 衝突により使用率に限界
B+Tree	ソート	SSD上	主記憶上インデックス不要	インデックスページの上書き操作が必要となり、SSDのライト性能が発揮できない

B+Tree 構造のように，SSD 上にインデックスを置く構造も考えられるが，インデックス情報の実体である Key は，Value より通常小さいと考えられ，この書換えのために SSD の上書き操作が余分に必要となることから，NAND Flash の性質からあまり適していないと考えられる．

ハッシュテーブル構造の課題は，最初にハッシュ関数が取り得る値を決めた時点で，サイズの上限が決まってしまい，スケーラビリティがない点である．また，HDD の場合には，ランダムアクセスが課題となる可能性があるが，4.2.3.1 の実験結果や，SSD 単体の実験結果(4.1.2)から，ランダムアクセスによる性能低下はほとんど見られず，この点は，Page-level FTL を使用していると考えられる，最新の SSD については問題がないものと考えられる．

第5章 Bloom Filter を利用したインデックス

5.1. Bloom Filter

ISAM方式を使えば，SSDにとって望ましいシーケンシャルライトと，参照の高速性を両立することが出来る．しかし，このために，主記憶上のハッシュテーブルを使用すると，巨大なテーブルが必要となってしまい，大容量のSSD上のデータを管理するために，大容量の主記憶が必要となってしまう．この問題を解決するためには，データの格納位置を保存するインデックスの構造を小さくする必要がある．このような特徴を持つインデックス構造として，B.Debnathら 2011 年 6 月に発表した関連研究のSkimpyStash[28]では，Bloom Filterをインデックスとして使用することを提案している．Bloom Filterは，”集合”を表現する”確率的”データ構造の一種であり，B.H.Bloomによって 1970 年に発表[32]され，Google Big Table[23]やhBase[24]でも使用されている．

ここで”集合”と呼んでいるデータ構造は，ある要素が，集合に属しているかどうかを判定する機能を持つデータ構造である．”集合”を表すデータ構造としては，例えばハッシュテーブルがあり，あるデータがその集合に属しているか否かを，確実に判定することができる．

Bloom Filter が”確率的”データ構造と呼ばれるのは，判定が確実でないためで，Bloom Filter で表現されたある集合について，実際に属している要素について”属していない”と誤判定することは決してないが，実際には属していない要素を”属している”と誤判定することはあり得る．つまり，偽陽性は有るが偽陰性はない．このような不確実なデータ構造は，他にメリットがなければ使われることはあり得ないが，Bloom Filter には，高速であり，メモリ消費量が，1%の偽陽性率を許容する場合に，9.6 ビット/要素と少ないというメリットがある．

5.2. Bloom Filterの基本的な動作

Bloom Filter の基本的な動作を図 32 に示す．Bloom Filter は，空の状態では，全てがゼロにクリアされた長いビット配列である．ここに要素を追加する場合，その要素についていくつかのハッシュ値を求め，それに対応するビットに 1 を立てる．要素が，集合に所属するかの判定は，同様にいくつかのハッシュ値を求め，それに対応するビットが全て 1 であるか否かで判断する．

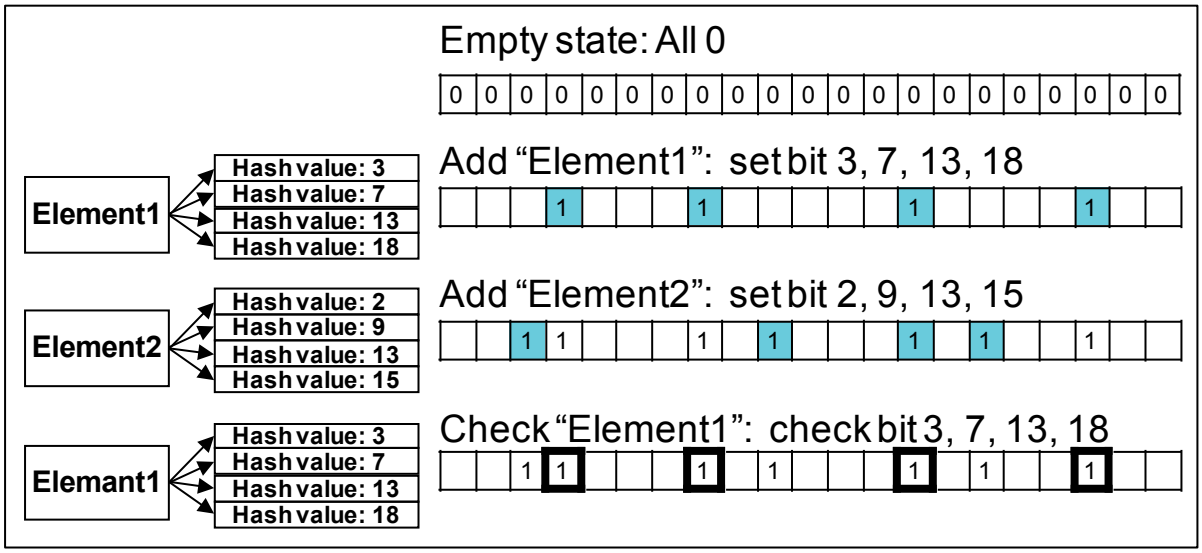


図 32. Bloom Filter の操作

従って，他の要素によって，対応するビット全てに偶然 1 が立っていれば，集合に属していると誤判定する偽陽性が発生する．

極端な例として，例えば，全てが 1 で埋まっているBloom Filterは，あらゆる要素に対して，所属していると誤判定する．一方で，集合に属していないと誤判定する偽陰性が発生することはない．Bloom Filterの配列に 1 が多く立っているほど偽陽性確率は上がるから，一つの要素に対して求めるハッシュ値の数が多いと，早めに，Bloom Filterが 1 で埋まってしまう．一方で，ハッシュ関数の数が少なすぎるとその全てに偶然 1 が立って偽陽性となる確率も上がる．このため，1 要素当たりのハッシュ関数の数には最適値があり，この最適値は式 1，この場合の偽陽性確率は式 2 の値となることが，既に示されている[33]．

m : Bloom Filterのビット数　n : 要素数
k : ハッシュ関数の数として、

$$k\text{の最適値} = \ln 2 \cdot \frac{m}{n} \approx \frac{9}{13} \cdot \frac{m}{n} \quad \text{(式1)}$$

$$\text{この時の偽陽性確率} \lambda = \left(\frac{1}{2}\right)^{k} \approx 0.6185^{\frac{m}{n}} \quad \text{(式2)}$$

5.3. hBaseでのBloom Filterの利用

Bloom Filter によるインデックス構造については，SSD 上の Key-Value Store について，SkimpyStash[28]で提案されているが，それ以前から，HDD 上の Key-Value Store では利用されていた．例えば，hBase[24]では，図 33 に示すように，MemStore 上で見つからないデータを，分散ファイルシステム上に保存されている hFile から順次探索する際に，高速化のために Bloom Filter を使用している．hBase では，Bloom Filter は，各 hFile の末尾に追記され，与えられたキーに対して各 hFile の Bloom Filter で順次所属判定を行って，どの hFile に与えられたキーが含まれているかを特定している．

単純な手法であるが，hFile の Bloom Filter は，各 hFile の一部であり，Bloom Filter の一致判定は，各 hFile に対するアクセス速度と比較して，十分高速であれば問題なく，実用になっているものと考えられる．

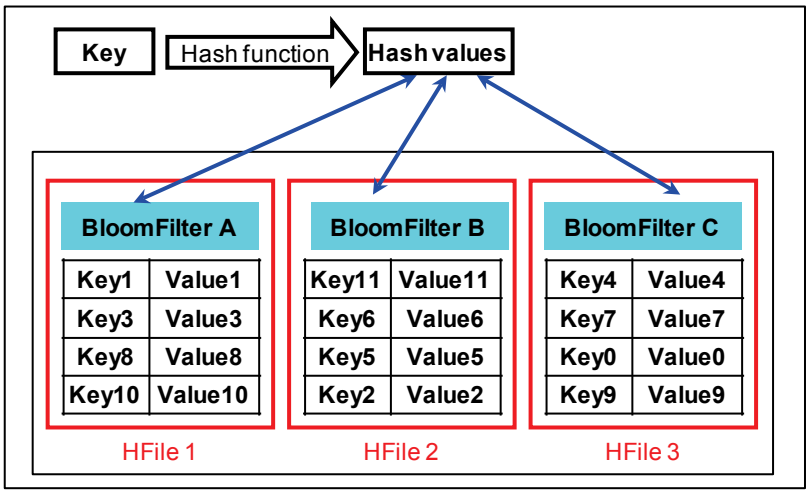


図 33. hFile 上の BloomFilter

5.4. 従来のBloom Filterインデックスの課題

5.4.1. 主記憶上の探索速度

本研究の目標の一つは，Bloom Filter の高いビット効率を利用して，高速ア

クセス性能を持つ SSD 上のデータ格納位置を，少ない主記憶容量で記憶する主記憶上インデックスを構成することである．従って，通常の計算機に実装出来る限りの最大の主記憶容量一杯，現在であれば少なくとも数 GB の主記憶一杯に Bloom Filter によるインデックスを配置した場合に，十分に高速に動作する必要があり，例えば，Bloom Filter が数 MB といった限定された容量の場合にだけ高速に動作する構造では，本研究の目的を十分に達することはできない．

ここで，Bloom Filter が，集合を示す構造で，関係を示すデータ構造ではないことが問題となる．関連研究のように，各ブロックに含まれる Key の集合を一つの Bloom Filter に記録する構造では，与えられた Key に対して，それが含まれるブロックを特定するために，例えば図 33 に示すように，各ブロックに対応した Bloom Filter に対して順次所属判定を行う必要がある．Bloom Filter の所属判定は，図 34 に示すように，Key から求めたいくつかのハッシュ値をアドレスとして，ビット配列上に 1 が立っているかどうかの検査によって実施される．通常，主記憶に 1 ビット単位でアクセスすることはできないため，1 ビットの検査であっても，CPU のキャッシュラインサイズで決まるサイズの読出しが行われ，必要のないビットも含めてキャッシュへの転送が行われることになる．

各ブロックの Bloom Filter について，それぞれ，ハッシュ関数の数だけのビットの読出しが行われることから，キャッシュラインサイズよりも十分にアドレスの間隔が広い場合には，概ね，ハッシュ関数の数×ブロック数に比例するメモリアクセスが発生することになる．

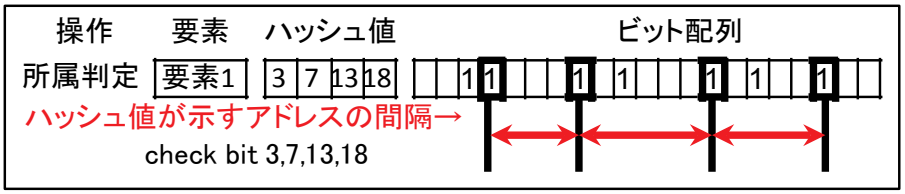


図 34．アドレスの平均間隔

要素当たりのハッシュ値の数を式 1 に示した最適値とすることを前提として，アドレスの平均間隔について計算すると，式 3 に示すように，1 つの Bloom Filter に含まれる要素数で決定されることになる．

m : Bloom Filterのビット数　n : 要素数

k : ハッシュ関数の数 $= \frac{9}{13} \cdot \frac{m}{n}$ とした場合

ハッシュ関数が示すアドレスの平均間隔lは、

$$l = \frac{m}{k} = \frac{m}{\frac{9}{13} \cdot \frac{m}{n}} = \frac{1}{\frac{9}{13n}} = \frac{13}{9} n \quad \text{(式3)}$$

このアドレスの平均間隔が，キャッシュラインサイズ以下では，Bloom Filter の全データを主記憶から読み出すことになる．この状態で，Bloom Filter によるインデックスが主記憶容量一杯に配置されているような状況を想定すると，インデックスで位置を決定するために，主記憶全体の読出しが必要となり，到底実用に堪えないことは明らかである．

現代の一般的な CPU のキャッシュラインサイズは 512bits(64B)程度である．この場合に，ハッシュ値が示す平均間隔がキャッシュラインサイズ以下となり，Bloom Filter の全てのビットがキャッシュに転送される条件を式 3 から計算すると，ブロック内の要素数:n≤354 となる．従って，1 ブロック当たりの要素数が 354

以下の場合，Bloom Filter による主記憶上インデックスは，大規模なデータに対して実用に堪えないと考えられる．

この効果を考慮して，Bloom Filter による主記憶インデックスのアクセス時間を計算した結果を図 35 に示す．例えば，1000M 要素，256 要素/ブロックでは，主記憶上インデックスのアクセス時間が 100ms に達し，実用的なインデックス構造とはいえない．

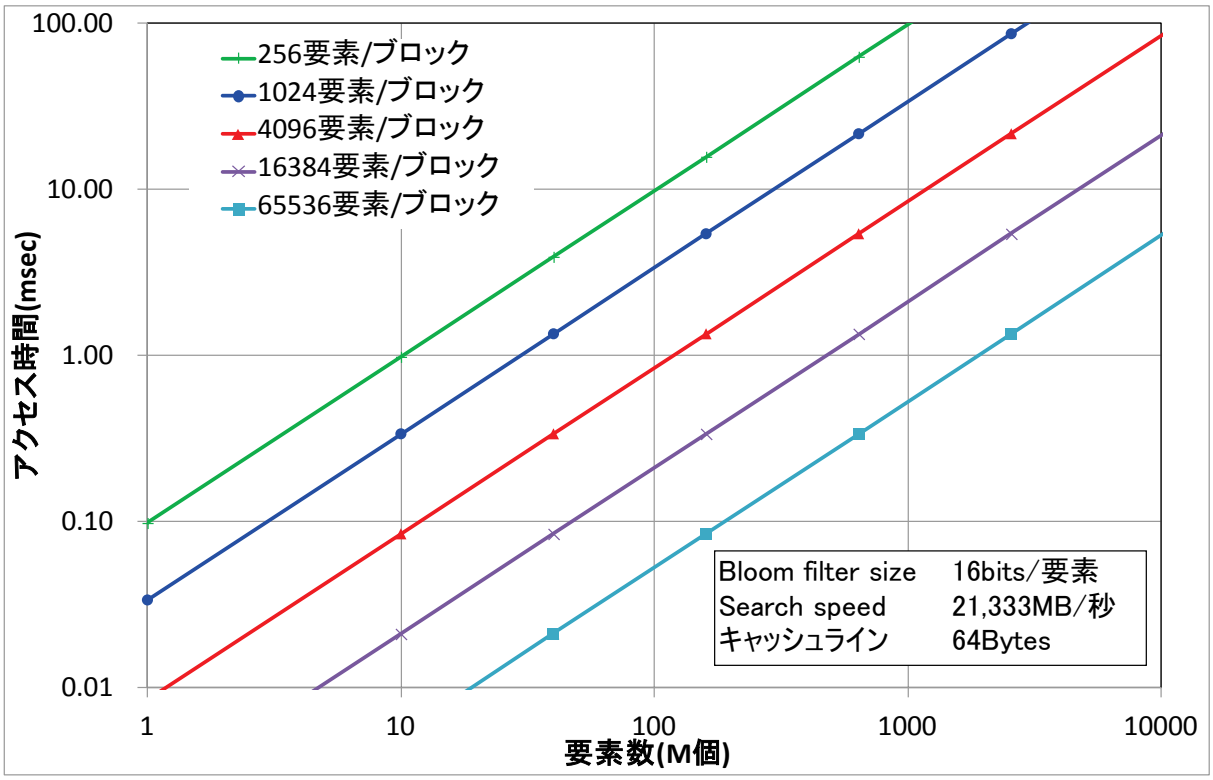
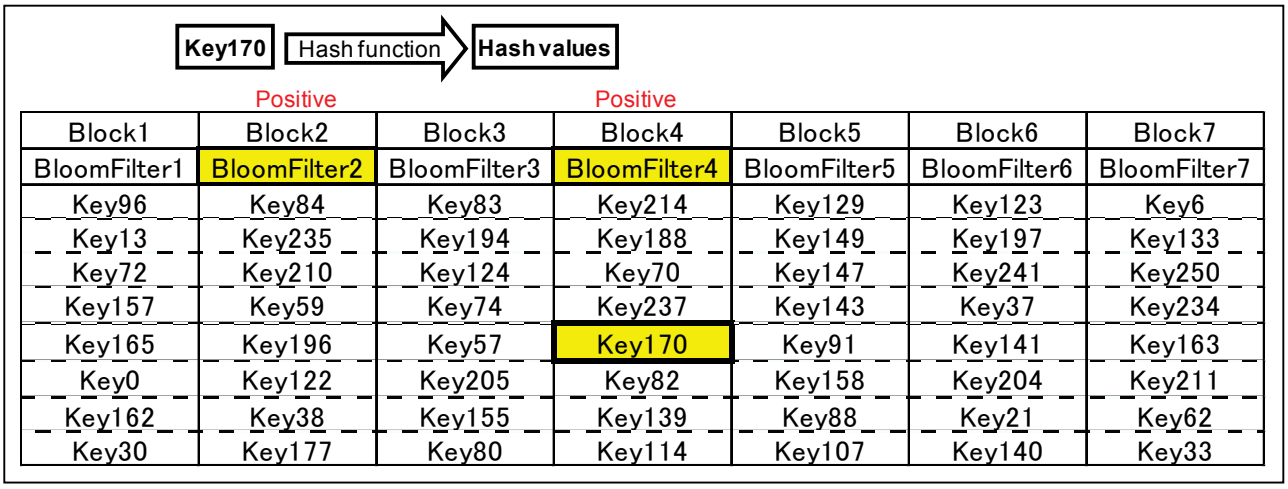


図 35. 要素数・ブロックサイズと主記憶インデックスアクセス時間

5.4.2. “空振り”による速度低下

B.Debnath らが既に指摘しているように，Bloom Filter をインデックス構造に使う場合，キーがない位置に誤って存在すると判定する偽陽性によって，”空振り”の読出しが発生し速度が低下するという課題がある．この”空振り”について，図 36 を参照して説明する．図 36 に示すように，8 個のブロックがあり，各ブロックに 8 個ずつのキーが格納されている状況を考える．各ブロックには，そのブロック内に含まれる全ての Key の集合を示す Bloom Filter がそれぞれ 1 つずつ対応している．

ここで．”Key170”に対してハッシュ値を求め，各ブロックに対応する Bloom Filter に対して所属判定を行ったとする．”Key170”が所属するのは Block4 で，Bloom Filter は偽陰性を示すことは決してないので，実際に所属する Block4 に対応する BloomFilter4 は陽性を返す．しかし，Bloom Filter は，偽陽性を示すことがあり，例えば，図 36 のように，Block2 に対応する BloomFilter2 が偽陽性を示したとすると，Block2 の中に”Key170”が存在しないことを確認する必要がある．これを”空振り”と呼んでおり，Block2 の中に”Key170”が存在しないことの確認に時間が掛かると，”空振り”による速度の低下が起こる．



Block1	Block2	Block3	Block4	Block5	Block6	Block7
BloomFilter1	BloomFilter2	BloomFilter3	BloomFilter4	BloomFilter5	BloomFilter6	BloomFilter7
Key96	Key84	Key83	Key214	Key129	Key123	Key6
Key13	Key235	Key194	Key188	Key149	Key197	Key133
Key72	Key210	Key124	Key70	Key147	Key241	Key250
Key157	Key59	Key74	Key237	Key143	Key37	Key234
Key165	Key196	Key57	Key170	Key91	Key141	Key163
Key0	Key122	Key205	Key82	Key158	Key204	Key211
Key162	Key38	Key155	Key139	Key88	Key21	Key62
Key30	Key177	Key80	Key114	Key107	Key140	Key33

図 36. Bloom Filter インデックスによる”空振り”

この”空振り”による速度低下について，定量的に検討する．従来手法では，ブロック内には，要素が到着順で追加されるため，ブロック内の要素の順番はランダムと考えられる．従って，ブロック内の探索に工夫の余地はなく，偽陽性で誤ったブロックを読みに行った場合，ブロック内に，その要素がないと確認するためには，ブロックサイズを N として，O(*N*)の時間が掛かると考えられる．

全体のブロックの中で k 個のブロックが偽陽性を示す確率はポアソン分布で近似できるから式 4 で与えられる．偽陽性によって，追加的に読み出す必要があるブロックのサイズは，全ブロックの読出し時間=1 として正規化した場合，k ブロックの読出し時間は k/M だから，結局，偽陽性ブロックの平均的な読出し時間は式 5 となる．

M:ブロック数　λ:偽陽性確率
全体の読出し時間＝1として、

$$\text{偽陽性ブロックが}k\text{個の確率} = \frac{\lambda^k e^{-\lambda}}{k!} \quad (式4)$$

$$\text{平均読出し時間} = \sum_{k=1}^{N-1} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \frac{k}{M} \quad (式5)$$

これをグラフにすると，図 37 となる．

1 要素当たりのビット数が 8，16 ビットの時の Bloom Filter の偽陽性率は，それぞれ約 2.14%，0.046%であるが，図 37 を見ると，全要素の読出し時間を 1 として正規化した平均読出し時間が，ブロック数が 2 では偽陽性率の半分，ブロック数が多い側では偽陽性率と等しくなっている．

これは，次のように説明できる．まず，ブロック数が 2 で有る場合，偽陽性が発生した場合に全体の 1/2 をスキャンする必要があることから，ブロック数 2 の場合，平均読出し時間は偽陽性率の 1/2 となる．また，十分にブロック数が多い場合は，全体のブロックの内，偽陽性の発生確率と同程度の比率のブロックを読出す必要があり，読出すブロックの比率は偽陽性率と等しくなる．

図 37 の結果からは，全体の要素数を何ブロックに分けて管理するか，言い方を変えれば，1 つのブロックあたりに含まれる要素数は，平均読出し時間で 2 倍の差しかもたらすことはなく，アクセス速度を決める本質的要素とはならないことがわかる．

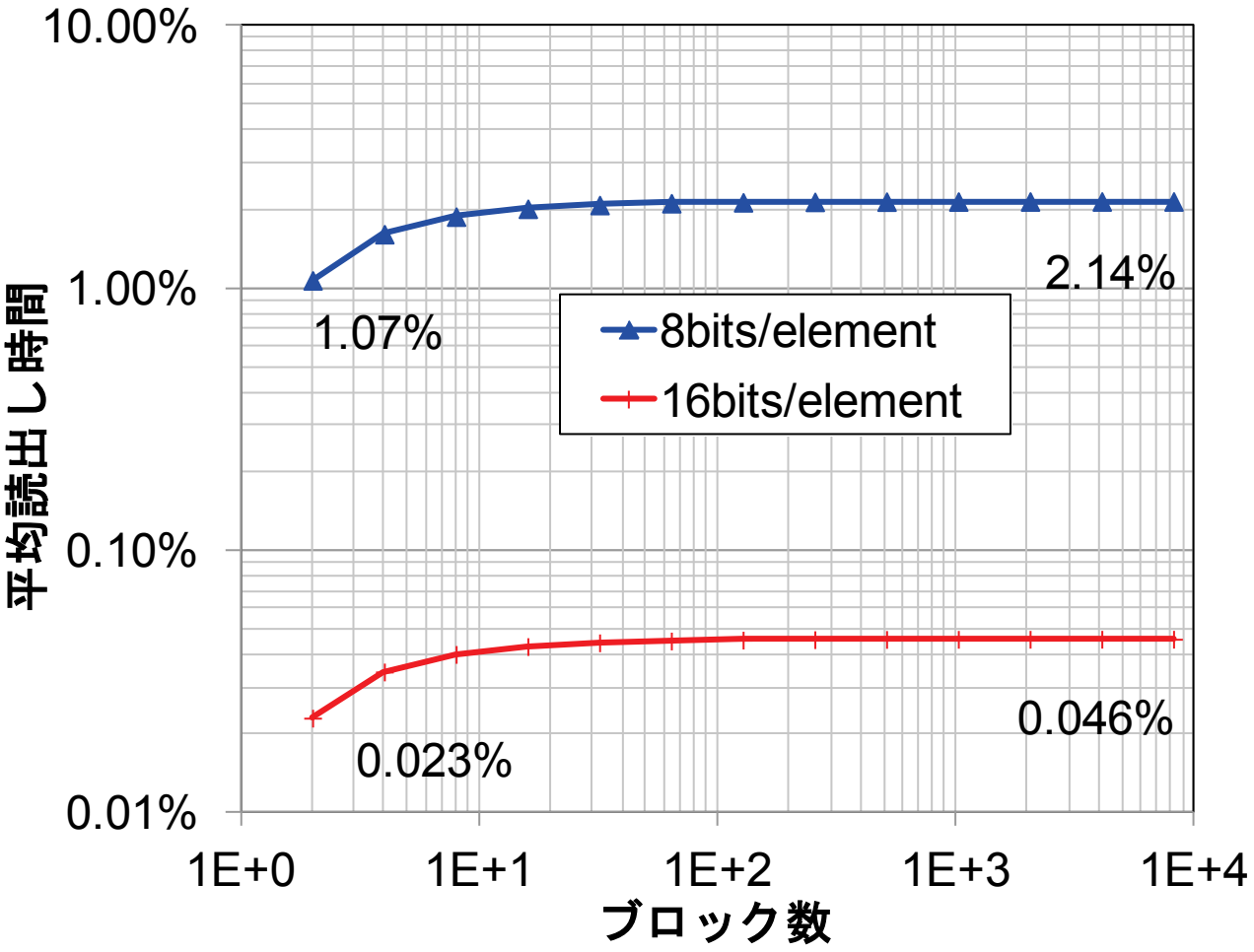


図 37. “空振り”に伴う平均読出し時間(全データ読出し=100%とする相対値)

第6章　本研究による提案

6.1. 主記憶上インデックス - Bloom Filter Map

主記憶上インデックスの探索高速化のために，本研究では図 38 に示す，Bloom Filter Map を提案する．Bloom Filter では，例えば，1%の偽陽性を許容する場合，必要なサイズは 9.6 ビット/要素で，ハッシュ値の数の最適値はこの 9/13 程度となる．本提案では，この数字は，Bloom Filter 全体のサイズ等には関係ないことを利用するため，Bloom Filter のビット配列の配置を工夫し，少ないメモリアクセスで，与えられた要素を含む Bloom Filter がどれであるのかを特定する．

具体的な構造を図 38 に示す．X 方向が，各ブロックに対応する Bloom Filter で，左端の数字は，ブロックの番号を示している．例えば，ブロック 3 に対応する Bloom Filter は，3 行目の”00110001010001010100”である．

今，“Key2”が，探索値として与えられ，そのハッシュ値が”2, 9, 13, 15”であったとすると，Bloom Filter Map では，それに対応する，Y 軸方向枠 4 ケ所を読みだす．これらのビット毎の AND を取って，1が立っているビットに対応するブロックが，与えられた探索値を含むブロックである．当然ながら，効率的なメモリアクセスのためには，この構造は，Y 軸の方向に局所性を持つように配置する必要がある．この構造は，ブロック数が増えると Y 軸方向，1 ブロック当たりの要素数又は Bloom Filter ビット数が増えると X 軸方向に拡張されていく．1 アクセス当たりのメモリ読出し量は，ハッシュ関数の数×ブロック数で概ね規定されることになる．

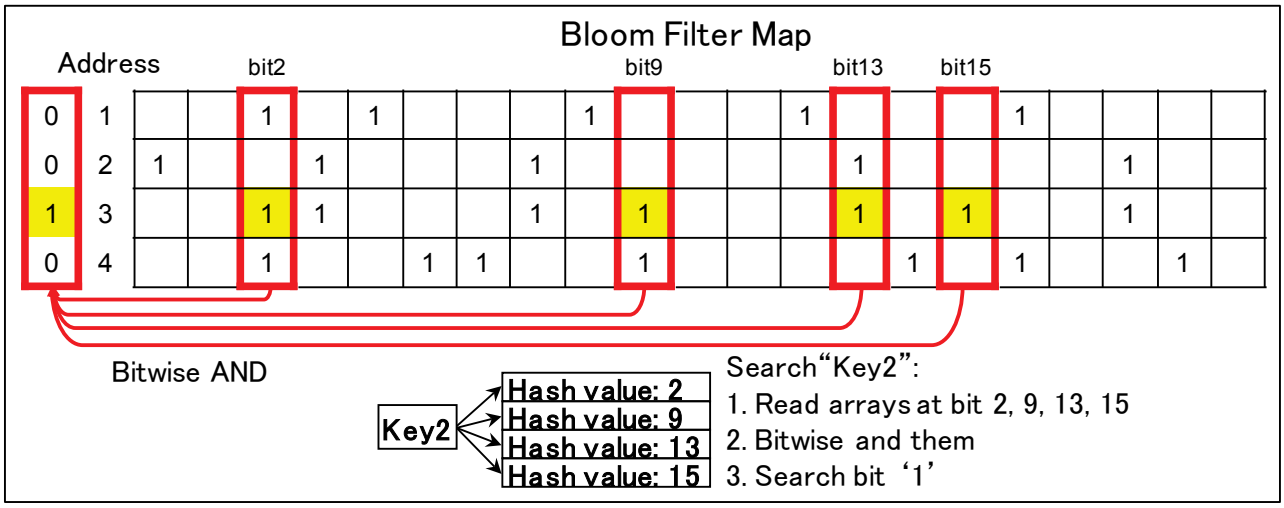


図 38. Bloom Filter Map

6.2. “空振り”対策 - 複合ブロック構造

Bloom Filter によるインデックスを使う限り，偽陽性の発生は避けられず，従って，実際には Key が存在しないブロックを，誤って探索してしまう”空振り”は避けられない．

また，5.4.2 で示したように，従来のデータ構造では，”空振り”時に，データの不存在を確認するために，ブロック内の全データの確認が必要であるためアクセス速度が低下していた．この場合，”空振り”時に読み出す必要があるデータの量は，Bloom Filter の偽陽性率のみで決定されるから，Bloom Filter のビット数を増やして偽陽性率を下げないと，性能が低下することになる．Bloom Filter のビット数を増やせば，当然，主記憶の消費量が増えてしまう．Bloom Filter のビ

ット数を増やさずにこの問題の解決するためには，”空振り”発生時に，ブロック内の全データを確認せずに，迅速に”空振り”を検知する必要がある．

この手段としては，以下の 2 通りが考えられる．

1. Key とブロック内の記録位置に規則性を持たせる
2. Key とブロック内の記録位置を別のインデックスに記録する

ここで，Bloom Filter では偽陽性は避けられないことから，重要なポイントとして，”存在するデータを迅速に見つけられる”だけではなく，”データの不存在を迅速に判定できる”必要がある．

1.の手法は，インデックス構造に記憶容量を取られないのが利点であるが，ランダムに到着する Key-Value 対に対して適用すると，ハッシュテーブルの衝突と同様の問題が発生し，充填率が約 2/3 程度で衝突確率が急激に増加し，追加・参照の時間が急激に増大する．従って，1.の構造は，ブロック内のデータが全て確定した後でないと利用することは難しい．2.の手法は，当然，インデックス分の記憶容量が必要となる．

これらの問題を解決するために，本研究では，図 39 に示す複合ブロック構造を提案する．到着したデータはとりあえず，到着順に非ソートブロックに順次記録し，Key と記録位置の対応は主記憶上のインデックスで管理する．非ソートブロックの数は限定されるため，主記憶上にインデックスを置いても，主記憶の消費は限定されるが，出来るだけ主記憶を節約するため，ここでも Bloom Filter Map を使用するものとする．以下，全体からどのブロックに Key が含まれているかを探索する Bloom Filter Map を”ブロック探索 Bloom Filter Map”，ブロック内でどのページに Key が含まれているかを探索する Bloom Filter Map を”ページ探索 Bloom Filter Map”と呼ぶことにする．

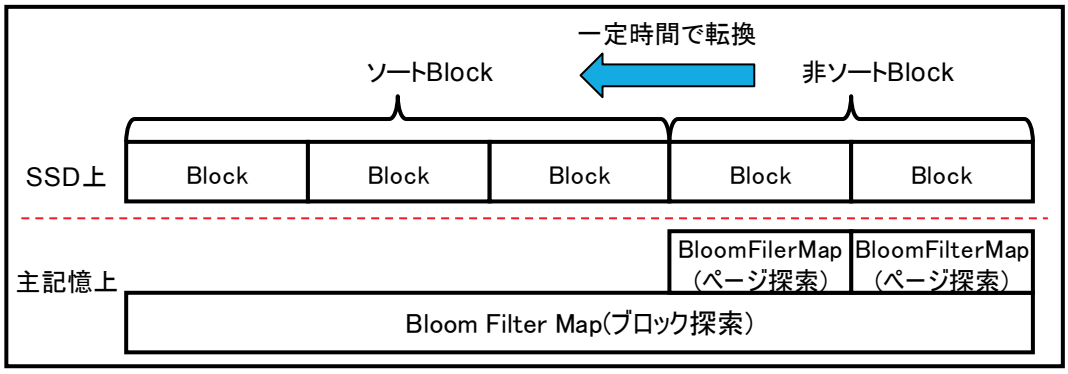


図 39. 複合ブロック構造

一定期間が経過したブロックは，Key のハッシュ値でソートしたソート済みブロックとして管理する．ソート済みブロックは，二分探索等によって効率的に探索することを想定しているが，KVS では，通常，Key-Value の長さは可変長であるので，Key-Value 対の境界が分かるように，ブロックより小さいページの単位で，図 40 に示すようにアライメントを取る．各ページのヘッダには，ページ内に含まれる Key のハッシュ値の上限/下限が記載されている．この構成で”空振り”が起きたブロックで，Key の不存在を確認するには，2 分探索により，Key が属しているページを特定した後，ページ内の全数探索が必要となるが，ページのサイズを SSD のアクセス単位に合わせておくと，通常のキャッシュアルゴリズムで先読みが行われるため，効率的なアクセスが可能となると考えられる．

Header	要素数	5	6	6	5	7	6
	min要素	5	1159	2223	13556	35559	40002
	max要素	1110	2309	3425	34456	38013	51145
要素領域		Element	Element	Element	Element	Element	Element
				Element			
			Element	Element		Element	Element
		Element			Element	Element	
			Element				Element
		Element	Element	Element	Element	Element	Element
		Element	Element		Element	Element	Element
						Element	
				Element		Element	
		Element	(blank)	Element	Element		(blank)

図 40. ブロックの内部構造

第7章　提案手法の評価

7.1. 主記憶上インデックスの評価

7.1.1. 評価条件

提案手法と, 従来の SkympyStash 等で使用されている Bloom Filter を順次探索する構造について評価した. ここで, 公平な評価のために, 各要素のアクセス頻度が一様分布となる場合だけでなく, latest/zipfian 分布による評価を加えている. zipfian 分布は, 出現確率 k 番目の要素の出現確率が 1/k であるようなアクセス分布であり, latest/zipfian 分布は, 更に要素の参照に"最近追加された要素ほど参照されやすい"といった偏りがある分布である. 順次探索を行う従来手法では, 確率が高い側から探索することで探索時間を短縮することができると考えられるため, latest/zipfian 分布の場合に有利である.

なお, これらのアクセス分布は, B. Cooper らが, Key-Value Store のベンチマークとして提案している Yahoo! Cloud Serving Benchmark (YCSB)[26]で用いられている分布を参考としたものである. B. Cooper らは, latest/zipfian 分布を, ブログ記事やニュース記事のように, 新しい情報ほど参照されやすい偏りを持ったデータのモデルとしている. また, 商品情報のように, 情報の新しさとは関係なくアクセス頻度に偏りがあるデータでは, アクセス分布は zipfian 分布になるとしている. zipfian 分布についても, アクセス頻度の偏りによって, 主記憶上の Bloom Filter がキャッシュにヒットする確率が上がる可能性があるため評価を行った. なお, 単純な zipfian 分布は, アクセス頻度の高い要素を優先的に探索することで, latest/zipfian 分布に変換することも比較的容易と考えられる. 詳細な評価環境を, 表 8 に示す.

表 8. 評価環境

CPU	Intel Core i5-2500 @3.30GHz(6MB L2 cache)
Chipset	Intel H67 Express
Main Memory	16GB(DDR3-1333M dual channel)
OS	Linux 2.6.35.11-83.fc14.x86_64
JRE	Open JDK IcedTea6 1.9.7
java version	1.6.0_20
Bloom Filter	16 bits/element
Hash function	11個
Block size	4096 elements/block

7.1.2. 評価結果

7.1.2.1. アクセスパターン毎のアクセス速度

主記憶上の探索時間について, アクセスパターン毎の評価結果を図 41, 図 42 及び図 43 に示す. 図 41 は, 各要素に対するアクセス頻度が一様に分布する場合, 図 42 はアクセス頻度が zipfian 分布である場合, 図 42 はアクセス頻度が zipfian 分布かつ「最近追加した要素ほど参照されやすい」方向性を持つ latest/zipfian 分布の場合である. 図 41 の一様分布及び図 42 の zipfian 分布では, 要素数の最も多い側で, 本提案は従来手法の 110.6 倍～112.1 倍高速であり, 図 43 に示す, latest/zipfian 分布でも, 11.8 倍の高速化を実現している. 図 43 の latest/zipfian 分布で差が縮まったのは, 確率の高い Bloom Filter から順次探索することによって, 順次探索的な従来手法が高速化するためである. 図 42 の方向性を持たない zipfian 分布でも, アクセスの偏りによって Bloom Filter がキャッシュに載りやすくなる効果が考えられたが, 実際には, 図

41 の一様分布の場合と大差なかった. 原因としては, 測定用のキーがキャッシュを占有することの影響などが考えられる.

7.1.2.2. ブロック当たりの要素数とアクセス速度

次に, ブロック当たりの要素数を 4096 個から変えて評価した結果を図 44 に示す. 結果は全て一様分布によるものである. 破線の囲みが従来手法, 実線の囲みが提案手法の結果である. 提案手法では, 速い側が 10us 程度で飽和しているものの, アクセス速度は, 全体の要素数には関係なくブロック数に概ね比例している. 主記憶上の探索時間を高速にするためには, ブロックサイズを出来るだけ大きくして, ブロック数を少なくすることが必要であることがわかる.

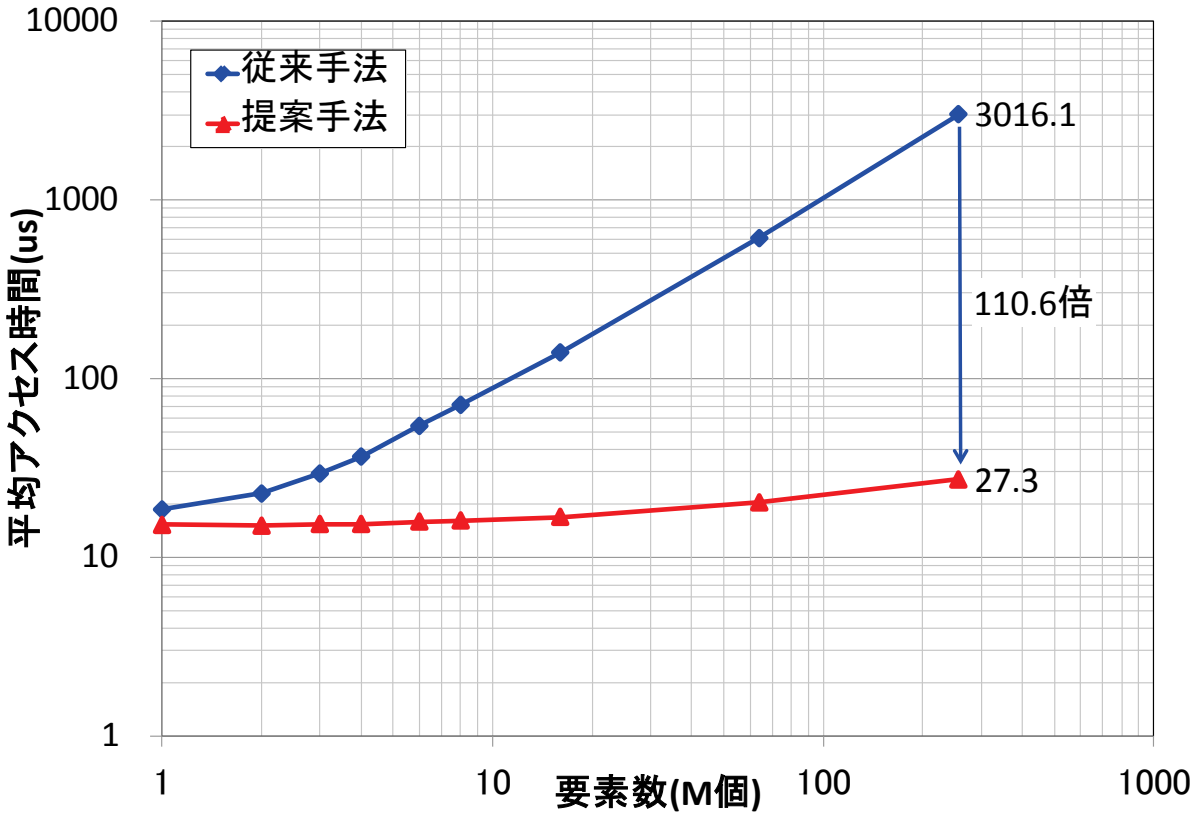

10000
1000
100
10
1
平均アクセス時間(us)
従来手法
提案手法
3016.1
110.6倍
27.3
1
10
100
1000
要素数(M個)

図 41. 一様分布

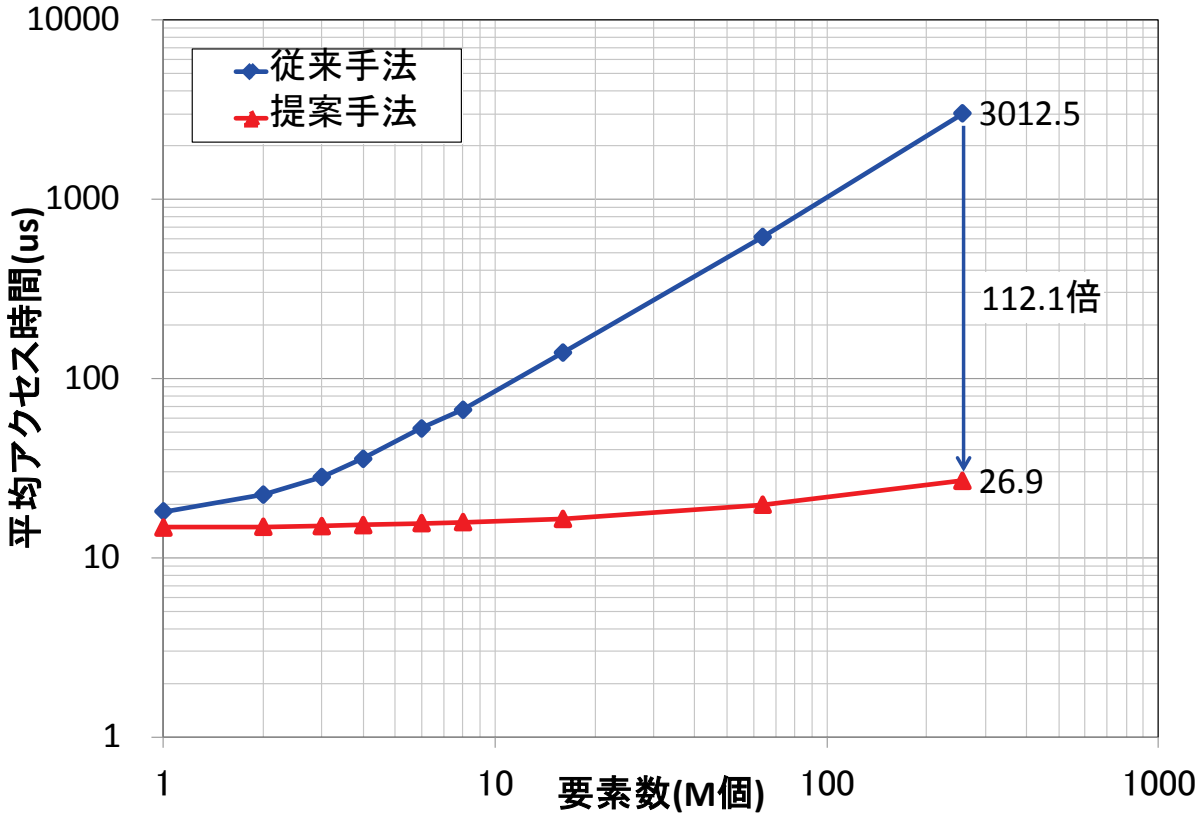


図 42. Zipfian 分布

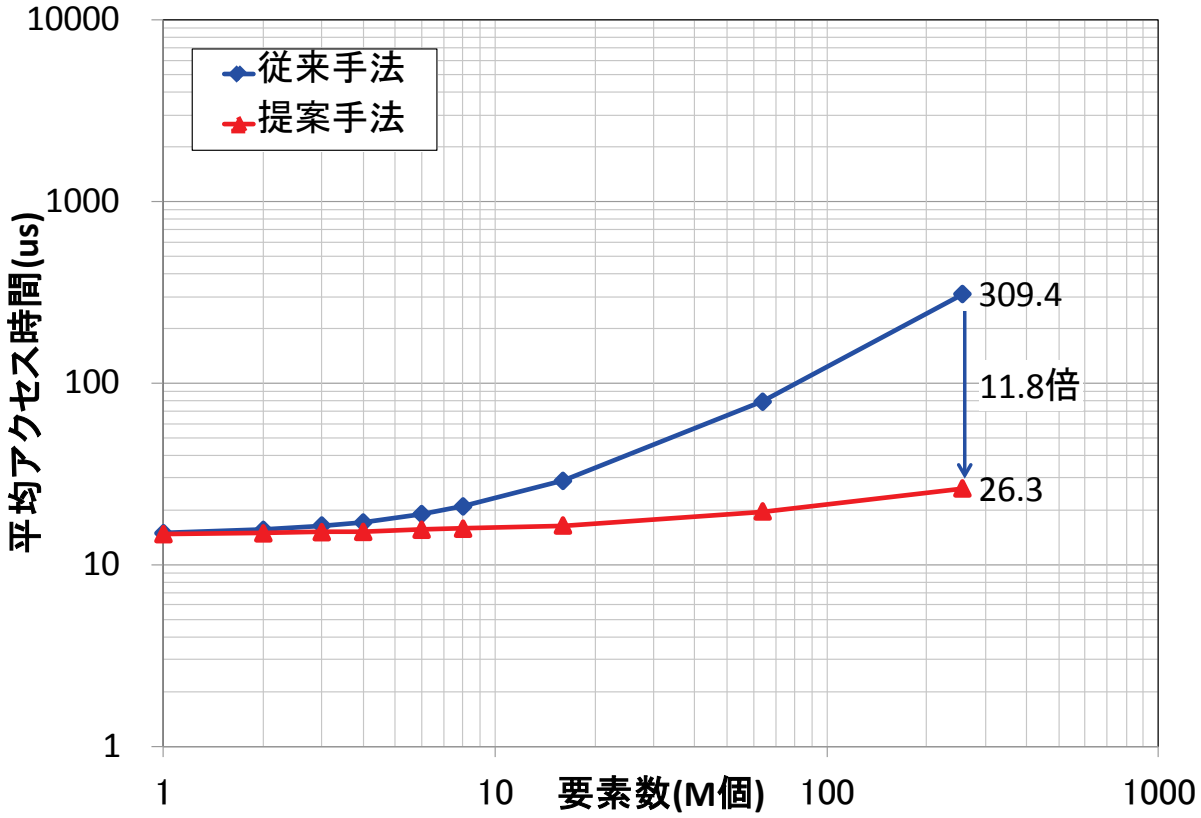


図 43. Latest/Zipfian 分布

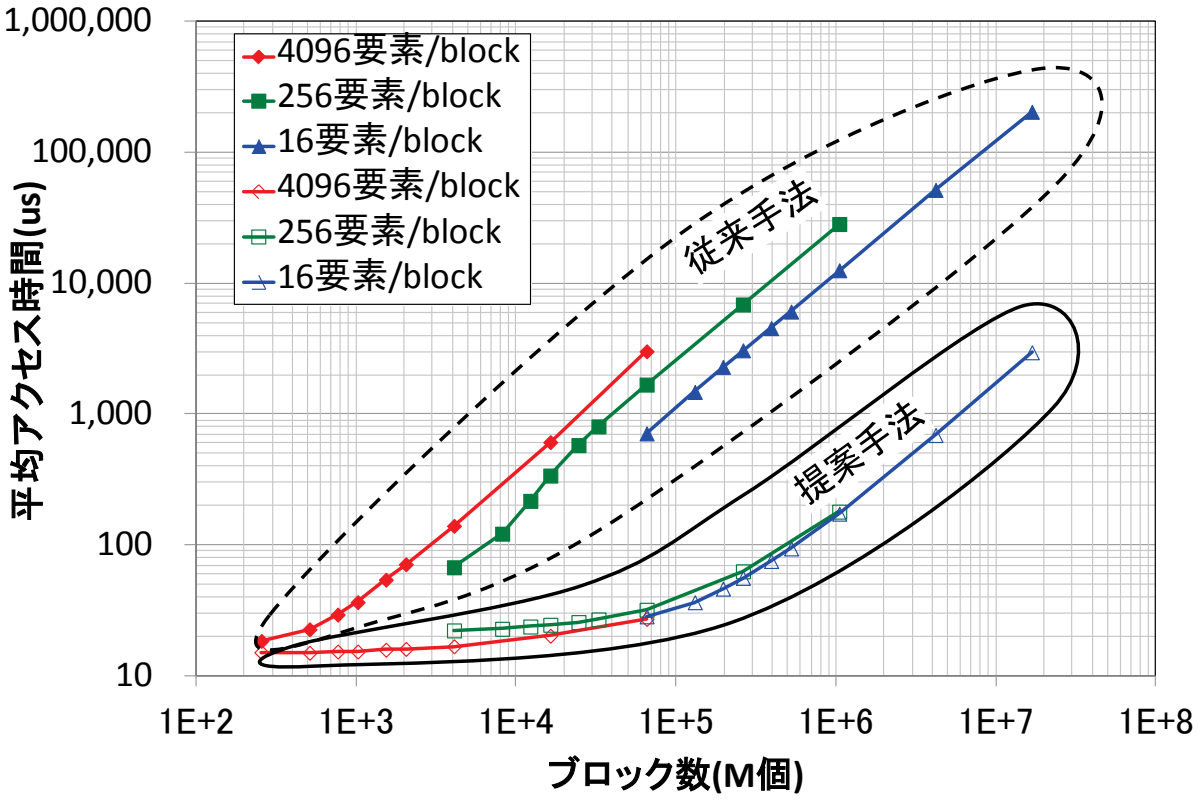


図 44. ブロック数対アクセス時間

7.2. 複合ブロック構造の評価

7.2.1. 評価条件

次に, 複合ブロック構造について検討するため, 評価環境に SSD を加え, ブロック構造毎のアクセス速度を評価した. 評価環境は表 8 と同様で, 評価に使用した SSD は, 4.1.2 で評価を行った intel X25-M(80GB)である. 各ブロックは, ファイルシステム上のファイルとして実装されている. その他の評価条件を表 9 に示す.

表 9. 評価条件

要素数	64M個
Keyサイズ	8Bytes
Valueサイズ	1000Bytes
ページサイズ	32768Bytes
ブロックサイズ	16MB～256MB
ブロック探索方式	Bloom Filter Map(16bits/要素)
ブロック内探索方式	Sort+2分探索, Bloom Filter Map(16, 24bits/要素)
ファイルシステム	ext4 with discard option

7.2.2. 評価結果

評価結果を図 45 に示す. X 軸がブロックサイズ, Y 軸がアクセス時間を示す. 各系列がブロック内のページ探索方式に対応しており, 非ソートブロックについてページ探索 Bloom Filter Map のビット数 16bits/要素, 24bits 要素の 2 通り, それに Sort+2 分探索を合わせた 3 通りである.

実際のシステムの特性はこの組み合わせとなり, ブロックサイズ 256MB 程度で, Sort+2 分探索と, 24bits/要素のページ探索 Bloom Filter Map を組み合わせれば, 概ね 1.5ms 以下程度の応答速度を持つ KVS が構築できることが分かる.

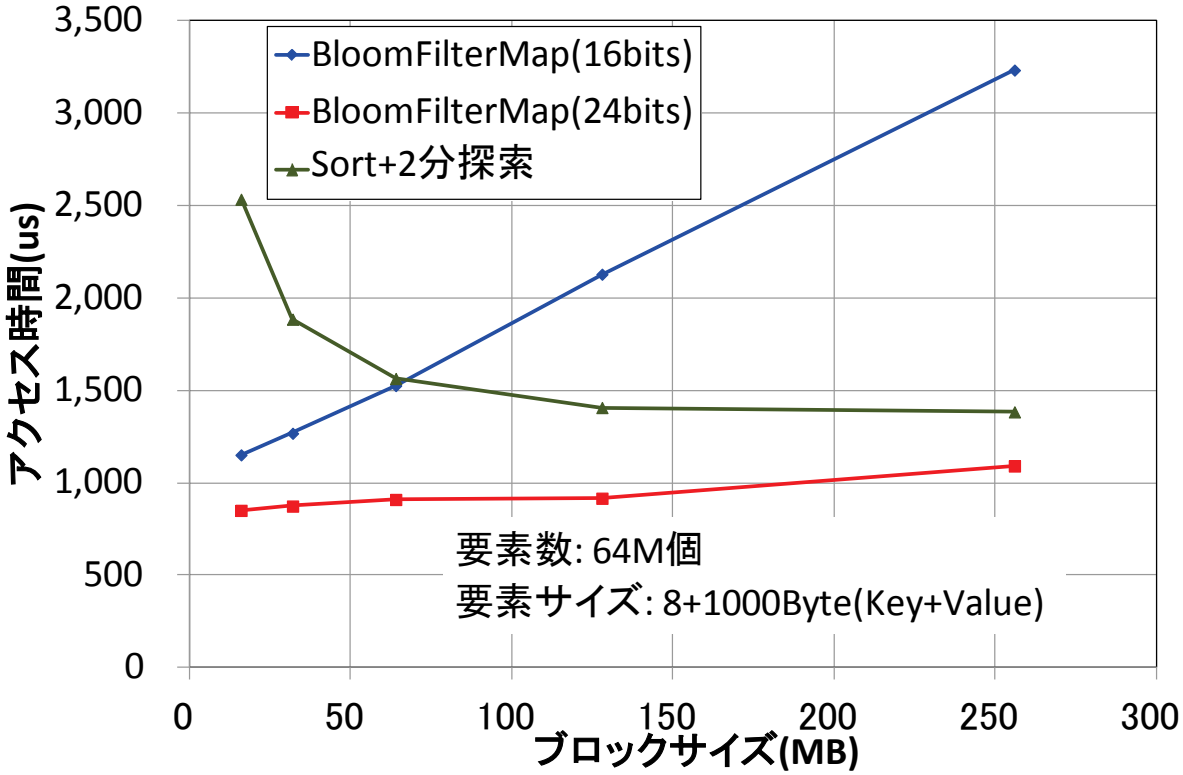


図 45. ブロックサイズ対アクセス時間

7.2.3. 考察

図 45 では, ページ探索 Bloom Filter Map を使う非ソートブロックでは, ブロックサイズにほぼ比例してアクセスタイムが長くなるのに対して, Sort+2 分探索では, ブロックサイズ大ほどアクセス速度が高速という, 全く逆の依存であった. この特性について検討する.

7.2.3.1. ブロックの探索回数

図 46 に, 平均ブロック探索回数のブロックサイズ依存性を示す. 図 46 の平均ブロック探索回数は, ブロック探索 Bloom Filter Map から提示された候補ブロック数で決定されるため, ブロック内のページ探索方式には依らない. ブロックサイズのみで決定され, ブロックサイズ大=ブロック数小ほど探索数は減少する. なお, 要素が見つかるとそこで探索を停止するため, 図 46 は, Bloom Filter の偽陽性率から予想される探索数よりは少ない数値となっている.

7.2.3.2. ページの探索回数

次に, 図 47 に平均ページ探索回数のブロックサイズ依存性を示す.

ブロック内のページ数は, ブロックサイズ大ほど増加する. ページサイズは 32KB 固定であるので, このブロックサイズの範囲では, ページ数は 512～8192 ページである.

本提案の複合ブロック構造では, ブロック内の全ページを探索するのではなく, ページ探索 Bloom Filter Map や, Sort+2 分探索によって, ページ探索回数を抑えることを狙っているが, いずれの方式でも, ページを全探索するのに比べるとページ探索回数を大幅に抑制している.

図 47 を詳細に見ると, Sort+2 分探索方式では, ブロックサイズ大=ブロック数小ほど探索数は減少しており, このグラフの形状は, 平均ブロック探索回数を示す図 46 とほぼ同じである. Sort+2 分探索方式では, ブロック探索 Bloom Filter Map が提示した陽性ブロックに対して, 最低 1 ページの探索が行われることから, 図 46 に示した平均ブロック探索回数とほぼ同等のグラフになるものと考えられる. なお, Sort+2 分探索では, ヘッダのみのページ探索も発生するが, この場合の探索時間は相対的に小さいと考えられたため, ここでは, ページ内全探索を行った回数をページ探索回数として示している.

次に, ページ探索 Bloom Filter Map 方式の 16bits/要素, 24bits/要素について見ると, 24bits/要素ではやや分かりにくいが, いずれも, ブロックサイズに比例してページ探索回数が増加している. これは, 以下のように説明できる. ページ探索 Bloom Filter Map 方式では, ブロック探索 Bloom Filter Map が提示した陽性ブロックが偽陽性であった場合, ページ探索 Bloom Filter Map が同じ Key に対して同様に偽陽性を示す確率は低いことから, 偽陽性ブロックの探索が行われる確率は低い. 従って, ブロック探索 Bloom Filter Map の偽陽性の影響を受けにくく, 常にほぼ 1 ブロックの探索が行われる. この場合の探索時間は, ページ探索回数で決まり, 探索するページ数は, ブロック内を探索するページ探索 Bloom Filter Map の偽陽性率で決まることから, ブロックサイズ大＝ページ数大側ほど, ページ探索回数が増加しているものと考えられる.

以上をまとめると, ソート済みブロックを Sort+2 分探索方式で探索する場合はブロックサイズ大ほどページ探索回数が少なくなって高速, 非ソートブロックを

ページ探索 Bloom Filter Map 方式で探索する場合はブロックサイズ小ほどページ探索回数が少なくなって高速であることが分かる.

このため, 全体を高速に保つためには, 例えば, ソート済みブロックについて Sort+2 分探索を高速化するためにブロックサイズを大きくしておきながら, 非ソートブロックについて十分なビット数のページ探索 Bloom Filter Map を割り当てるといった対策が必要となる.

7.2.3.3. ページの探索回数とアクセス時間

次に, ページ探索回数とアクセス時間の関係について検討する.

図 48 にページ探索回数と, アクセス時間の大部分を占めるページ探索時間の関係を示す. ほぼ線形な関係となっており, アクセス時間は, ほぼページ探索回数で決定されていることが分かる.

また, この結果では, 1 ページ探索あたりの探索時間が 1100us～1400us 程度掛かっているが, SSD からの 1 ページ 32KB の読出し時間は, 表 3 に示した SSD のスペックからは 200us 程度, 図 18 及び図 19 に示した筆者による測定でも 177us を得ており, ページ内の探索時間には, 改善の余地があると考えられる.

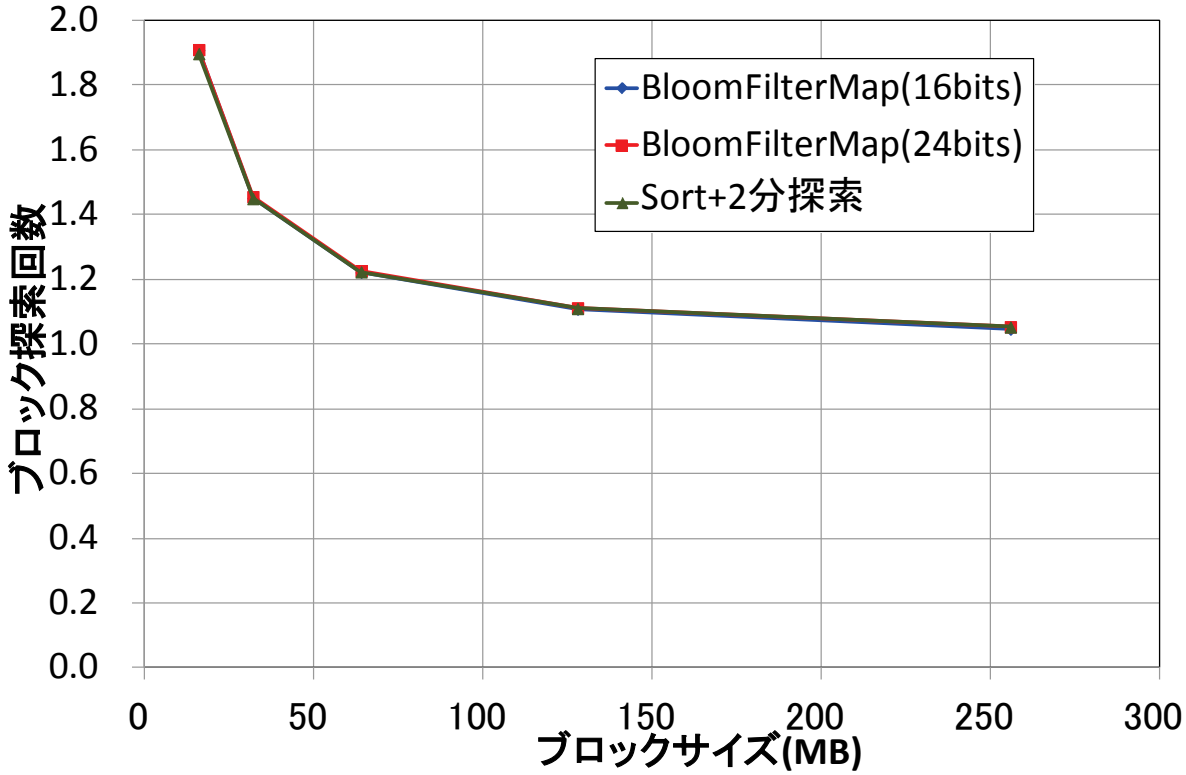


図 46. ブロックサイズ対ブロック探索回数

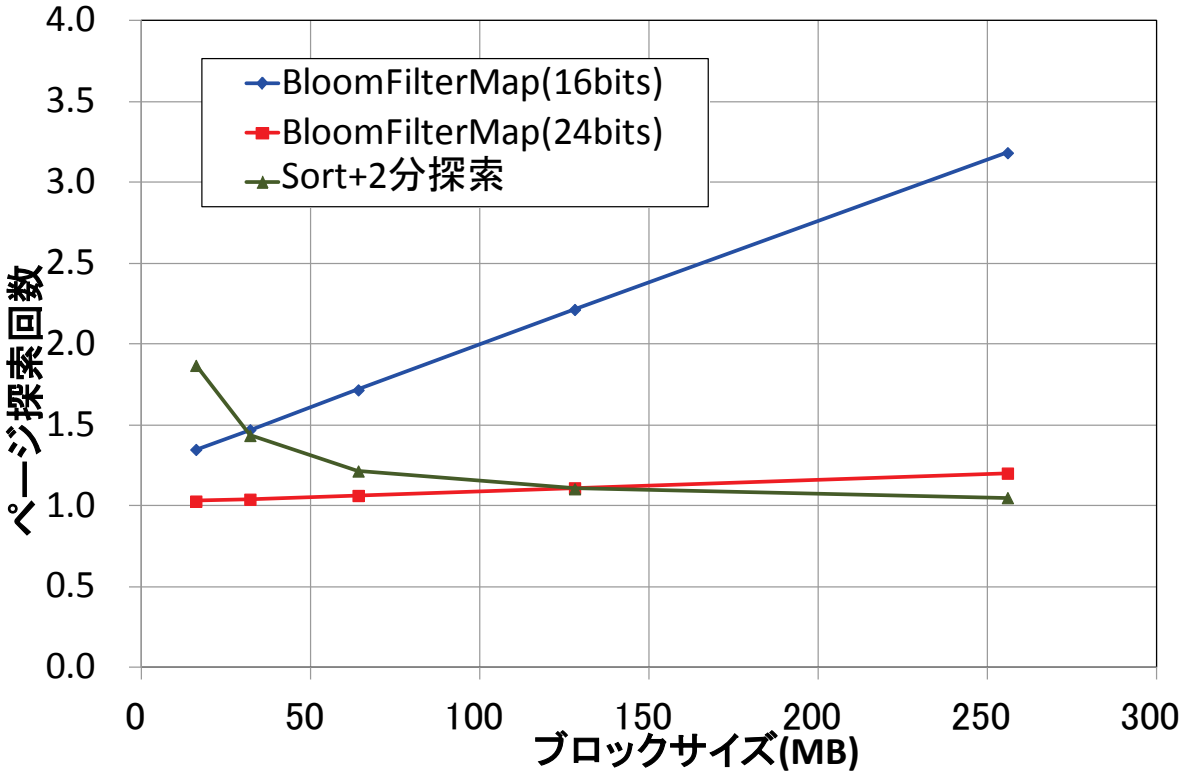


図 47. ブロックサイズ対ページ探索回数

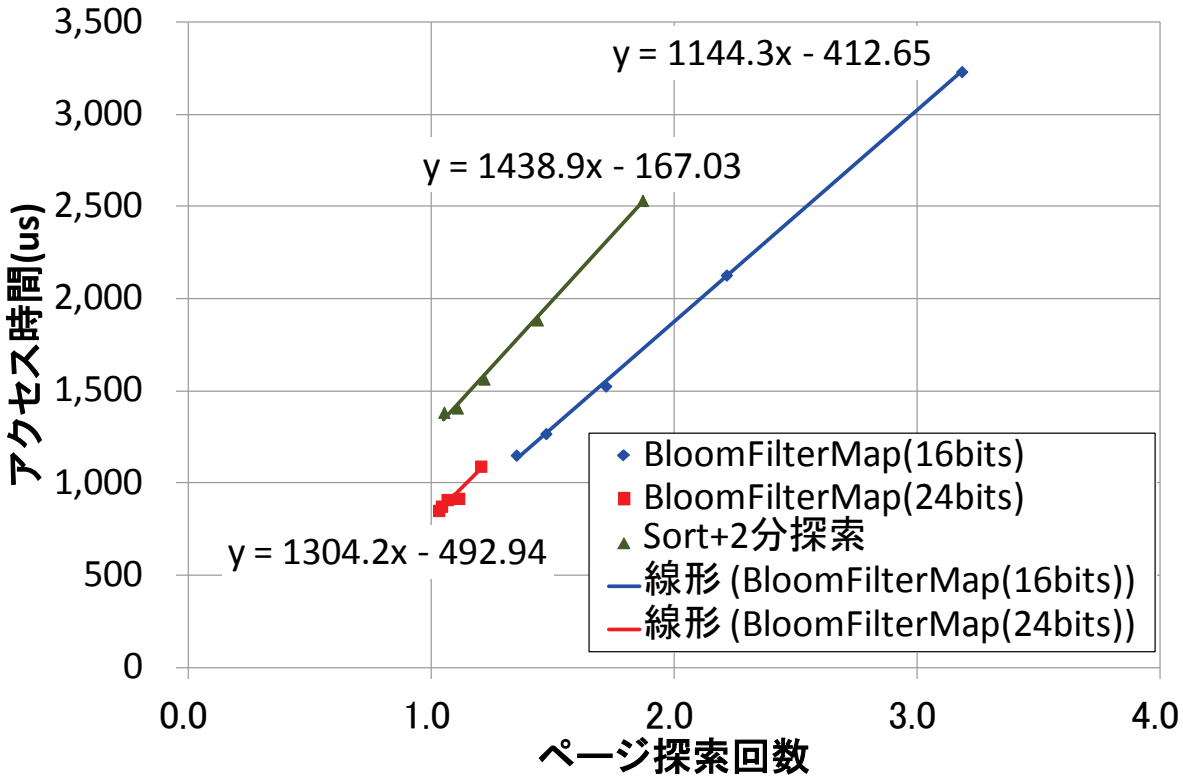


図 48. ページ探索回数対アクセス時間

第8章　システム上での利用形態

8.1. システム上での利用形態

SSDを対象としたKey-Value Storeをシステムで実際に利用する場合，主記憶とSSDの容量比をどの程度に取るかが一つのキーパラメーターとなると考えられる．この最適値を，本質的な容量単価から推定される経済的制約，最大容量に関する技術的制約，及び需要面から推定し，本研究の妥当性について検討する．

8.2. 経済的制約

図 5 に示したように，NAND Flash メモリと DRAM のメモリセル面積は，NAND Flashメモリが3bit/セルのMLC技術を使う場合に，概ね2:9の比率となる．この面積比が単位容量当たりのコストの比率を示しているとして，SSD 容量当たりの DRAM+SSD の合計価格を計算した結果を図 49 に示す．図 49 の X 軸は，SSD/DRAM の容量比であり，当然，容量比が拡大し，割安な SSD の比率が増加するほど SSD 容量当たりのシステム単価は低下する．この前提条件では，SSD 容量当たりの”DRAM+SSD 価格”の低下は，SSD/DRAM 容量比 10 倍程度で飽和しており，最低でも SSD/DRAM の容量比は 10 倍程度とすることが経済的制約からは必要であると考えられる．実際のシステムコストには，”DRAM+SSD 価格”以外の価格も追加されるため，SSD 容量当たりのシステム単価が飽和するのは，更に SSD/DRAM 容量比が大きい場合である．

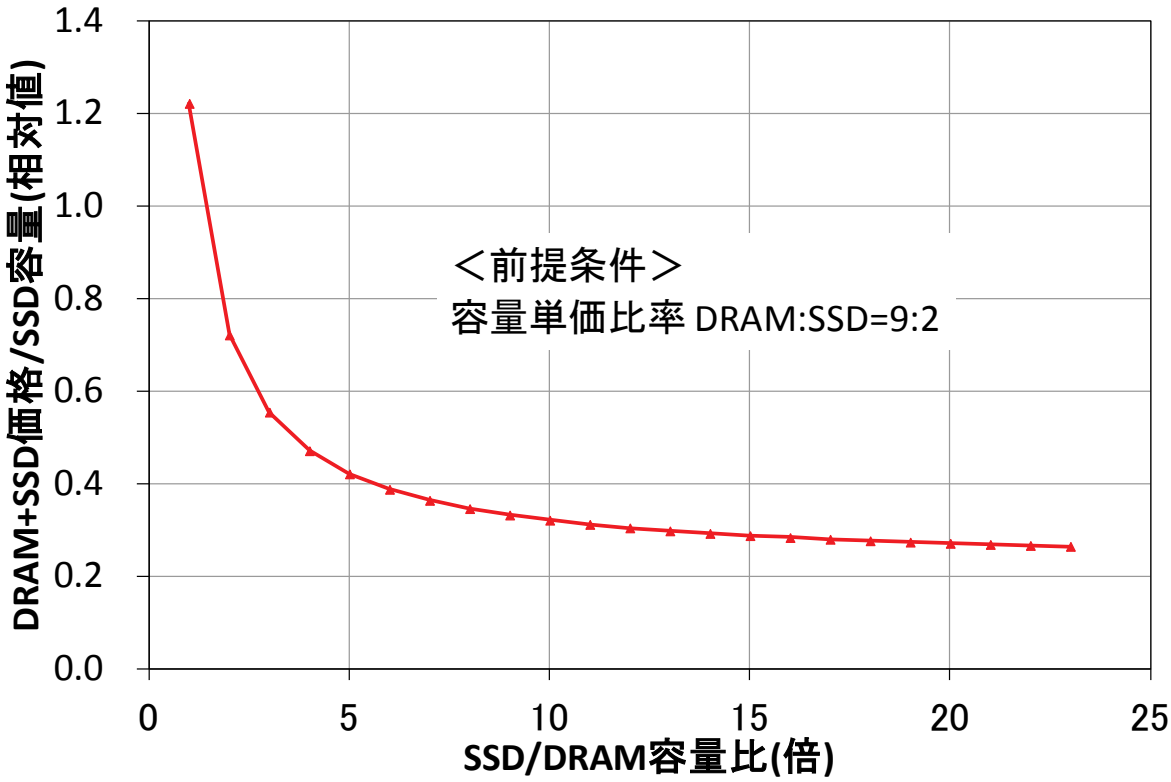


図 49. SSD 容量当たりの DRAM+SSD 合計価格

8.3. 主記憶のスケーラビリティによる制約

主記憶として使用されるDRAMは，通常，極めて高速なバス上に接続されていることから，従来から，スケーラビリティに制約があることが指摘されて来た．一例を挙げると，J. Haasらは，高速バスに多数のメモリモジュールを接続することは技術的に難しく，バス速度の向上とともに，1 チャネル当たりのメモリモジュール

接続数は減少し，800Mbpsでは，バスの分岐が困難となり，システム当たりのメモリモジュール接続数を増やすことが困難になると推定していた[34]．これに対して，NAND Flashメモリは，SSDの形態で使われる限りは，通常，Serial ATAインターフェースに接続されて使われるため，Serial ATAインターフェースの数が問題となる．具体的に，最新の，サーバー向けの 2CPUマザーボード[35]で，接続可能な容量を比較する．ここで，DRAMメモリモジュールの最大容量は 8GB，SSDの最大容量はintelが販売している最大容量である 600GBとして計算すると，SSD:DRAMの容量比は 37.5 倍となる．

当然ながら，システム当たりの接続ポート数は，システムの設計によって変動すると考えられる．しかし，DRAMの接続ポート数については，先に述べたように，高速バスのスケーラビリティに課題があることから，ポートの増加は比較的困難である．これに対して，SSDのポートについては，例えば，SATAインターフェースについては，Port Multiplier[36]等によって，大幅に増加させることが可能である．従って，上述のSSD:DRAM容量比 37.5 倍も下限であり，さらなる拡大も可能であると考えられる．

表 10．システムあたりの最大接続容量

種類	Interface	ポート数	容量/装置(GB)	容量/システム(GB)
DRAM	DDR3-DIMM	12	8	96
SSD	SATA	6	600	3600

8.4. KVSで必要とされる容量比

これに対して，KVS の需要面から，必要とされる容量比を検討する．

表 11 は，16bit/要素の Bloom Filter Map を使用する場合に必要な SSD:DRAM の容量比を，Key+Value サイズ毎に計算した結果である．ここで，1 要素のサイズは，Key+Value サイズ+要素長などを格納する 8B とした．

表 11．平均 Key+Value サイズと SSD/DRAM 容量比

平均Key+Valueサイズ	SSD/DRAM容量比
16	12
64	36
256	132
1024	516
4096	2052

KVS が，どの程度の Key+Value サイズで使用されるかは必ずしも明らかではないが，例えば，Yahoo! Cloud Serving Benchmark (YCSB)[26]では，1 つの Key に対して，合計 1000B の Value が紐付けされるとして評価しており，必要な SSD:DRAM 容量比は約 500 倍となる．これは，経済的制約による 10 倍以上，主記憶スケーラビリティから推測される 37.5 倍以上と比べると大きい比であり，現在の典型的な設計のハードウェアでは，主記憶容量に余裕が出来る可能性が高いと言える．

しかし，このことは，Key-Value Store の利用形態によって，主記憶容量の余裕に差が生まれることも意味する．平均 Key+Value サイズが想定より小さく，SSD の容量に対して，想定より大量の要素が追加された場合，主記憶は不足する．また，本提案の複合ブロック構造では，未ソートブロック内のインデックスを主記憶上の Bloom Filter Map に頼っているが，この Bloom Filter Map も，ブロック当たりの要素数が多い場合にはサイズを拡大しないと，偽陽性率が高くなってパフォーマンスを低下させる．

第9章　複合ブロック構造の改良

9.1. 複合ブロック構造の課題

7.2 で示したように，複合ブロック構造は，当初の期待通りの動作を実現できたが，約 1.5ms 程度と，SSD そのものの速度に比べれば遅く，改良の余地があると考えられた．このため，アルゴリズム上，実装上の改良を行った．

9.2. 改良ハッシュテーブル方式

4.1 に測定結果を示したように，採用した SSD では，Page-level FTL を使用していると考えられ，ランダムライトとシーケンシャルライトによるアクセス速度の差は認められない．また，実際に，キーのハッシュ値でブロック内の書込み位置を決定するハッシュテーブル方式による測定結果(図 31)でも，キーの衝突が発生する充填率約 2/3 以下の領域では，速度の低下は見られない．従って，充填率の問題さえ解決できれば，ブロック内データ構造として利用できる可能性がある．

この問題を解決する方法として，図 50 に示す改良ハッシュテーブル方式を提案する．改良ハッシュテーブル方式では，キーから得られたハッシュ値に応じて，ブロック内のいずれかのページに要素を登録するハッシュテーブル方式と，先に示した複合ブロック構造で利用したページ探索 Bloom Filter Map を複合的に利用する構造であり，衝突を発生することなくブロック内の所定のページに書き込むことが出来た要素については，Bloom Filter Map に登録せず，衝突が発生し，他のページに振り替えた要素のみを Bloom Filter Map に登録する．

平均的には約 2/3 のデータは衝突を発生しないため，Bloom Filter Map に必要とされる容量は，全ての要素を Bloom Filter Map に登録する場合に比べて約 1/3 で済むことになる．

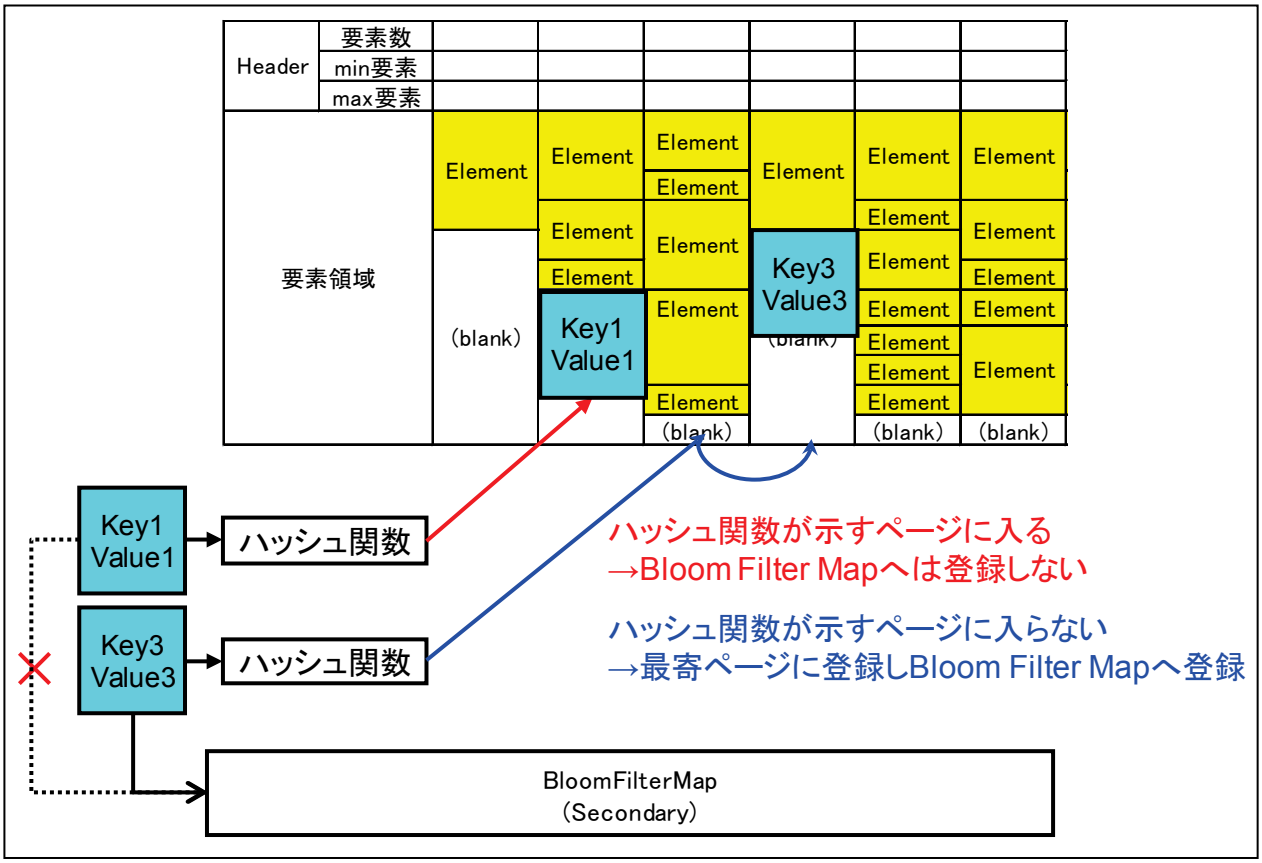


図 50. 改良ハッシュテーブル方式

また，図 50 に示す改良ハッシュテーブル方式は，主記憶の空き容量や，

Bloom Filter Map の埋まり具合に応じた主記憶利用の柔軟性を提供することもできる. 改良ハッシュテーブル方式では, 主記憶容量が不足したり, Bloom Filter Map が想定より速く埋まったりしている場合には, ブロックを完全に要素を埋めずに途中でブロックへの追加を終了することも考えられる. 改良ハッシュテーブル方式によれば, その場合でも, 通常のハッシュテーブルと同様に約 2/3 程度の充填率を確保することが出来る.

9.3. ページインデックスの集中配置

従来のソート済みブロックでは図 40 に示すように, 各ページの先頭に, そのページ内に含まれるキーのハッシュ値の範囲が記録されており, これを 2 分探索していた. しかし, キーのハッシュ値の範囲は, SSD の最小読出し単位を大きく下回る int 型×2 要素の 8 バイトであり, 分散して配置されていることから, アクセス効率が下がっていたと考えられる. この問題を解決するために, 各ページのヘッダ部分を, ブロックの先頭に集めアクセス効率の向上を図った.

9.4. 先読みによるアクセス効率の改善

システムによるファイルキャッシュを期待して, プログラム側でのキャッシュは行っていなかったが, 実際には, ファイルキャッシュの効果はさほど明確ではなかった. このため, 必ず, 1 ページをまとめて読み出すように実装上の変更を行った.

この変更と, 9.3 ページインデックスの集中配置によって, 複合ブロック構造の内, ハッシュテーブル方式では 1 ページ, Sort+2 分探索方式では 2 ページの読出しでアクセスが完了することが期待される.

第10章 改良方式の評価

10.1. 評価条件

改良方式について, 7.2 複合ブロック構造の評価と同じ条件で評価を行った.

10.2. 評価結果

評価結果を, 図 51 に示す. X 軸がブロックサイズ, Y 軸がアクセス時間を示す. 各系列がブロック内のページ探索方式に対応しており, 改良ハッシュテーブル方式について, 衝突要素を管理するページ探索 Bloom Filter Map のビット数 16bits/要素, 24bits 要素の 2 通り, それに Sort+2 分探索を合わせた 3 通りである.

アクセス時間について見ると, いずれも, ブロックサイズ大ほどアクセス速度が高速という特性になっている. 実際のシステムの特性はこの組み合わせとなり, ブロックサイズ 256MB 程度で, Sort+2 分探索と, 16bits/要素の改良ハッシュテーブル方式を組み合わせれば, 概ね 800us 以下程度の応答速度を持つ KVS が構築できることが分かる. これは, 図 47 の改良前と比べて約 2 倍の速度である.

10.3. 考察

ブロックサイズ対ページ探索回数(図 51)のグラフを改良前(図 47)と比較すると, Sort+2 分探索は単に高速化されただけである. しかし, ページ探索 Bloom Filter Map で管理されたブロックと比較して, 改良ハッシュテーブル方式のブロックの特性が大きく違っている. 違いとして, 以下の 2 点が指摘出来る.

1. Sort+2 分探索方式と同様にブロックサイズ大側が高速となった
2. ページ探索 Bloom Filter Map のサイズによる差が無い

この特性は, 空振り発生時の挙動の変化で, 以下のように説明出来る.

改良ハッシュテーブル方式では, ページ探索 Bloom Filter Map に対して Key 追加が行われる確率が衝突時のみで 1/3 程度となったことから, ページ探索 Bloom Filter Map の偽陽性確率は大きく減少したと考えられる. 一方で, ページ探索 Bloom Filter Map にはブロック内の全要素が記録されているわけではないため, 偽陽性であっても, ハッシュ値が指し示すページに要素が記録されていることの確認が必ず必要となった.

従って, ブロック探索 Bloom Filter Map の偽陽性発生時には, 改良ハッシュテーブル方式では, 必ず 1 ページの読出しが発生する. この時, ページ探索 Bloom Filter Map も同じ Key に対して偽陽性を示すと, 更に 1 ページの読出しが発生するがこの確率は低い. よって, 読出し時間は, 概ね, ブロック探索 Bloom Filter Map が偽陽性を含めて陽性と提示した各ブロックについて, 各 1 ページを読出す時間になると考えられる.

実際に, ブロック探索回数とアクセス時間の関係をグラフにした結果を図 52 に示す. ほぼ比例しており, 上記の仮説が正しいことが確認できる. なお, ここでは, 要素が発見された時点で探索を打ち切っているため, 確認を行ったブロック数は, Bloom Filter Map が偽陽性を含めて陽性と提示したブロック数よりは少なくなっている.

また, 改良前の方式では, ソート済みブロックに対して行われる Sort+2 分探索と, 未ソートブロックの探索では, ブロックサイズに対するアクセス時間の特性が

逆依存となっていたため，ブロックサイズの最適値を検討する必要があった．これに対して，改良後の特性では，いずれも，ブロックサイズ大側ほど高速となったため，ブロックサイズは出来るだけ大きくすれば良く，パラメーター調整の負担が減ったものと考えられる．また，図 53 に示すように，1 ページ当たりのアクセス時間も 400us と高速になっており，実装上の改良も確認できた．なお，ここで，Sort+2 分探索時に 1 ページ当たりのアクセス時間が長いのは，集中配置されたインデックスの読出し時間を含むためと考えられる．

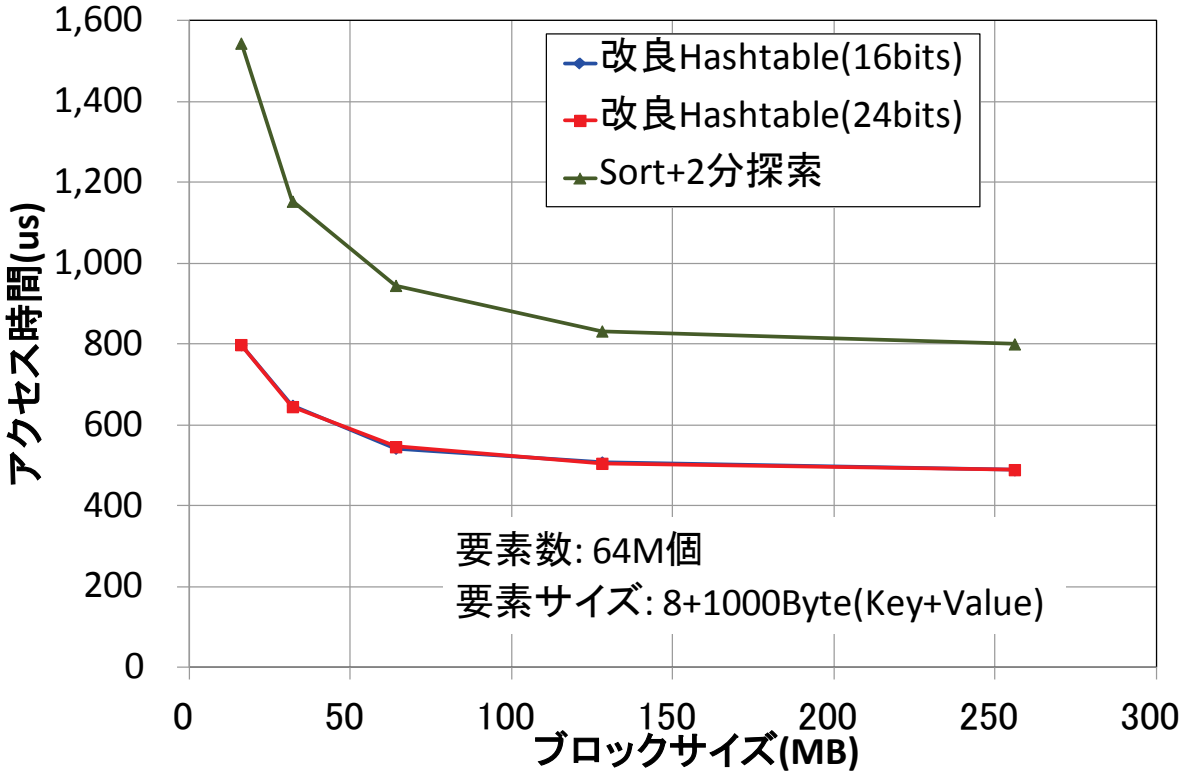


図 51. ブロックサイズ対アクセス時間

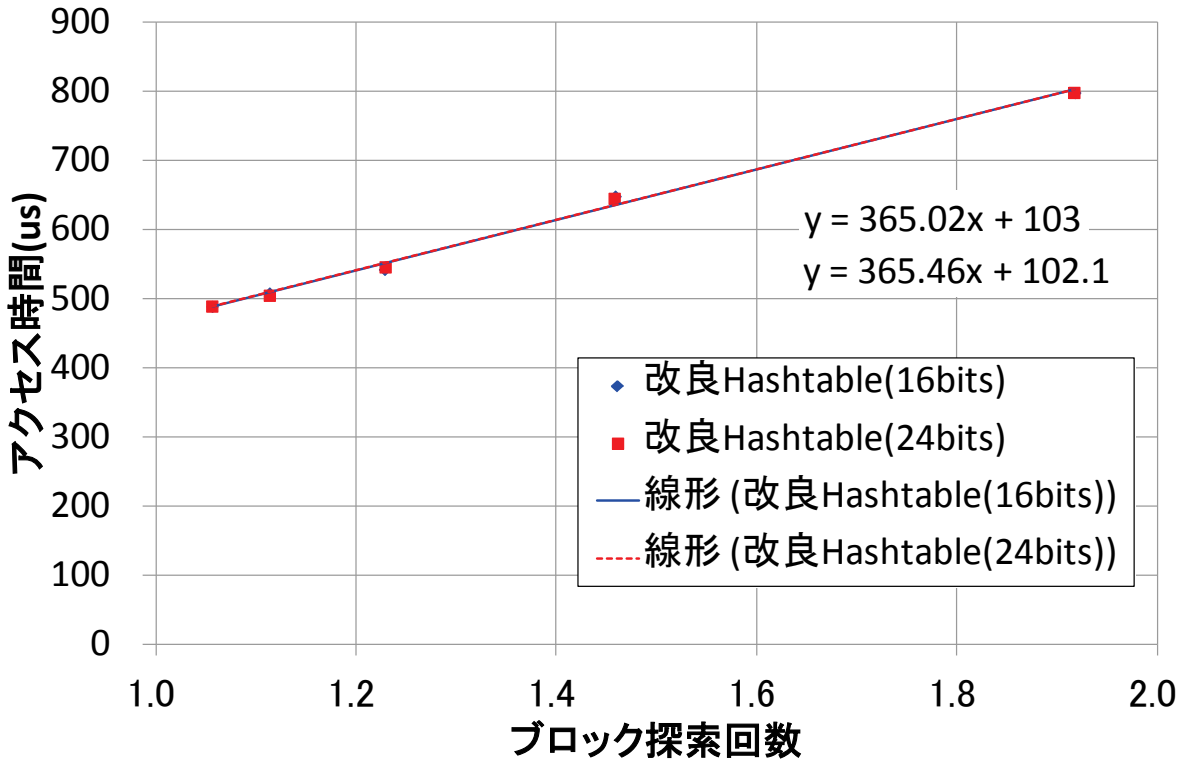


図 52. ブロック探索回数対アクセス時間

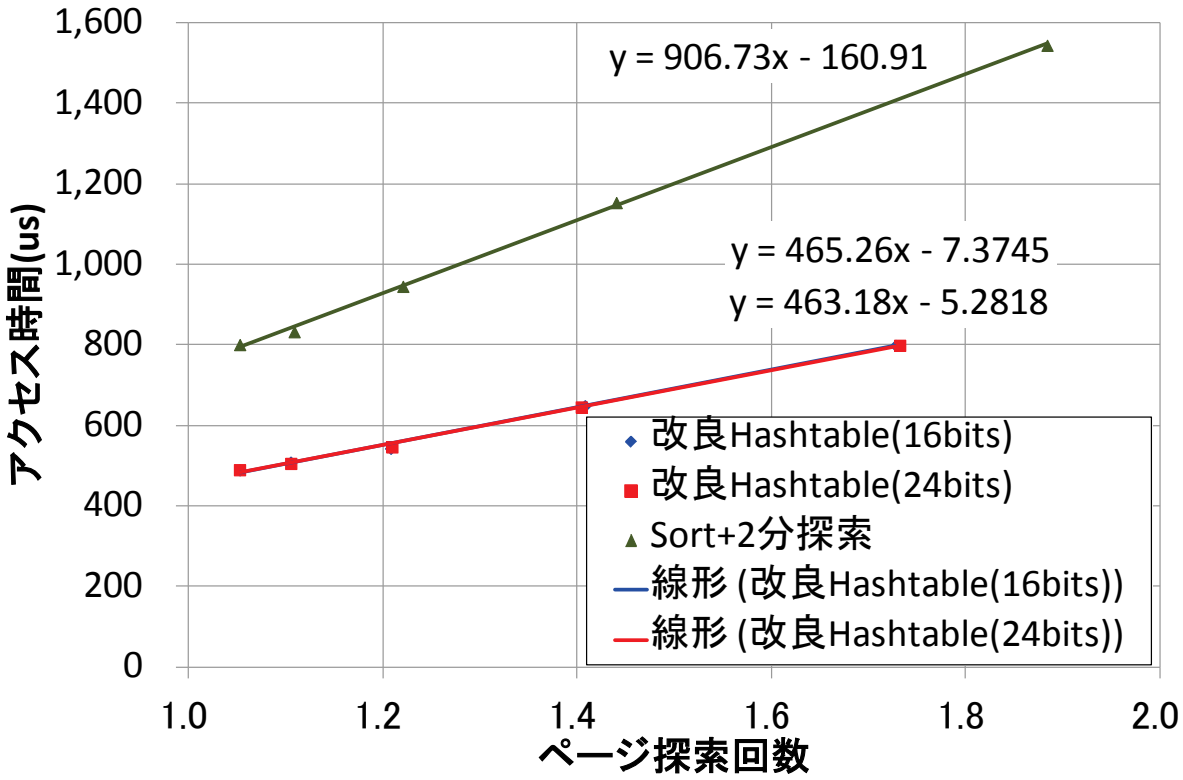


図 53. ページ探索回数対アクセス時間

第11章 結論

11.1. まとめ

本論文では, NAND Flash メモリによる SSD 上の Key-Value Store について検討を行った. まず, NAND Flash メモリと, それを利用した SSD の性質について検討を行った. 関連研究[17]が指摘するように, SSD で見られるランダムライトの速度低下には, 従来, メモリカードを中心に利用されていた Block-level FTL(Flash Translation Layer)に起因するものと, ページの上書きに起因するものの 2 種類が存在する. 本論文の予備検討による SSD 及び USB フラッシュメモリの測定では, 前者の Block-level FTL に起因する速度低下は, USB フラッシュメモリでは依然として観察されるものの, 外部 DRAM バッファを使用した最新の SSD では見られず, Page-level FTL が使用されている可能性が高いことを明らかにした.

従来, SSD 上の Key-Value Store のデータ構造としては, Key-Value 対をシーケンシャルに書込み, 主記憶上インデックスを用いて高速にアクセスを行ういわゆる ISAM 方式が適しているとされてきたが, この結果から, ページの上書きが発生しない前提であれば, キーの値で記録位置を決定するハッシュテーブル的な手法も利用可能であることを明らかになった.

また, 従来の ISAM 方式を利用する関連研究についてさらに検討を進め, 主記憶上インデックスとして, 高速でメモリ消費量の多いハッシュテーブルか, 低速でメモリ消費量の少ない Bloom Filter のいずれかが利用されており, 高速性とメモリ消費量の低減の両立には至っていないことを指摘した.

この内, メモリ消費量を低減する観点から, Bloom Filter を利用したインデックス構造を主な研究対象として選択し, Bloom Filter を利用したインデックス構造の低速性の原因を検討し, 以下の 2 点が原因であることを明らかにした.

第一の原因は, ビットを単位としたデータ構造である Bloom Filter に順次アクセスするために必要なメモリアクセスコストであり, その影響を定量的に見積もった. この対策として, 主記憶からの読出しを最低限の量とする Bloom Filter Map 構造を提案し, 主記憶上インデックスの速度では 11.8 倍を実現した.

また, 第二の原因として, Bloom Filter の偽陽性による”空振り”が, 速度低下の要因となることは従来から指摘されていたが, この空振りによる速度低下は, ブロックのサイズには依らず, Bloom Filter の要素当たりビット数のみで決定されることを明らかにし, Bloom Filter のビット数を増やさずに高速化をするためには, ブロック内のデータ構造の工夫が必須であることを明らかにした. この対策として, ブロック内インデックスを使う未ソートブロックと, ソート済みブロックを併用する複合ブロック構造を提案し, 評価を行った. この結果, 一般消費者向けの安価な SSD を使用して, 1.5ms 以下のアクセス速度を実現する見通しを得た.

この速度は, SSD 単体の速度に対して不十分と考えられたことから, 更にデータ構造上, 実装上の改良を行った. その結果, 非ソートブロックについて, 衝突要素の記録位置のみを Bloom Filter Map で管理する改良ハッシュテーブルを使う方式に採用し, その他実装上の改良も進めた結果, 同じ SSD を使用して 800us 以下のアクセス速度を実現する見通しを得た.

本研究の途中, 修士 2 年中間発表(2011 年 5 月)の後に, 比較的近い研究として, Bloom Filter を主記憶上インデックスとした SSD 上の Key-Value Store である SkimpyStash[28]が発表されている. しかし, この研究を前提としても, Bloom Filter による主記憶上インデックスを利用して, 高速な Key-Value

Store を構築するために必要なデータ構造について進んだ検討を行っており，本研究には意義があったものと考える．

また，SkimpyStash を含め，従来の研究では，HDD，SSD の性能上の問題から，要素をシーケンシャルライトする方式について主に検討がなされていたが，Page-level FTL を使用する SSD を採用する場合には，上書きを避けるという前提条件のもとでは，ランダムライトを行うデータ構造も利用可能であることを明らかにした点も，本研究の意義である．

11.2. 今後の展開

本研究では，ランダムライトを行うデータ構造についても，改良方式として検討を行い，一定の成果を得ている．しかし，この分野については，従来の検討が少ないため更に工夫の余地があると考えられる．また，本研究では，最新の SSD が Page-level FTL を採用しているということを前提条件としたが，Page-level FTL を採用する SSD が一般化するという確信が無い限り，これ以上の進んだ研究は困難である可能性もある．この点については，SSD の製造業者の協力が必要であり，それを含めた研究が推進されることを期待したい．

また，本研究の実装では，さほど本質的要件と考えられなかったことから，要素のアップデートについては全く考慮されていない．上書きが困難である SSD の特徴を前提と知れば，hBase で使用されているように，要素をタイムスタンプ付きで記録し，列挙した中で最新の値を返すといった工夫が必要になると考えられる．この場合，例えば，最新の 3 バージョンのみ保持といった動作を実現するためには，ブロックのソートを行うタイミングで，必要以上に古い要素の削除を行うといった工夫も必要になると考えられる．

謝辞

本研究を進めるにあたり，日頃より親身の御指導と，適切なご助言を頂いた，修士論文指導教官の浅野正一郎教授に感謝いたします．中間発表の場に置いて，また，個別に時間を取って，本研究に関しましてご討論頂きました，浅見徹教授，喜連川優教授をはじめとした，東京大学大学院の先生方に深謝申し上げます．さらに，日頃より，研究室の環境整備，事務処理等で御協力を賜りました，跡部香秘書に心から感謝いたします．修士課程の学生生活においては，浅野研究室の Mohammad Kamrul Islam さん，安達研究室の木村光樹さん，鈴木貴敦さん，他の学生諸氏にも様々な御協力を頂きました．深く感謝いたします．

また，在職のまま修士課程に進学するにあたっては，エルピーダメモリ株式会社の各位にも休暇の取得等に御協力を賜りました．ここに深く感謝いたします．

発表文献

山田淳二,浅野正一郎, “Bloom Filterによる高速でコンパクトなインデックス構造 ～ NAND Flash SSD上の高速データベースを対象とした ～”, 信学技報, vol. 111, no. 255, CPSY2011-30, pp. 31-36, 2011 年 10 月.

參考文獻

[1] R. Micheloni, A.Marelli and S. Commodaro, “NAND overview: from memory to systems”, Inside NAND Flash Memories, chapter 1-2, Springer, USA, October 2010

[2] Takahashi, T.; Sekiguchi, T.; Takemura, R.; Narui, S.; Fujisawa, H.; Miyatake, S.; Morino, M.; Arai, K.; Yamada, S.; Shukuri, S.; Nakamura, M.; Tadaki, Y.; Kajigaya, K.; Kimura, K.; Itoh, K.; , "A multigigabit DRAM technology with 6F2 open-bitline cell, distributed overdriven sensing, and stacked-flash fuse," Solid-State Circuits, IEEE Journal of , vol.36, no.11, pp.1721-1727, Nov 2001

[3] Hynix semiconductor datasheet HY27UK08BGFM 32Gbit (4Gx8bit) NAND Flash 2007/2/9

[4] The CompactFlash Association http://compactflash.org/

[5] SD Association https://www.sdcard.org/home/

[6] http://en.siliconmotion.com/A3.2_Partnumber_Detail.php?sn=6

[7] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. “A space-efficient flash translation layer for compact flash systems,” IEEE Transactions on Consumer Electronics, vol.48, no. 2, pp. 366-375, 2002.

[8] http://www.kernel.org/doc/Documentation/filesystems/ext4.txt

[9] Feng Chen , David A. Koufaty , Xiaodong Zhang, Understanding intrinsic characteristics and system implications of flash memory based solid state drives, Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, June 15-19, 2009, Seattle, WA, USA

[10] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In Proc. of ASPLOS’09, 2009.

[11] OLTP Trace from UMass Trace Repository. OLTP Trace from UMass Trace Repository. http://traces.cs.umass.edu/index.php/Storage/Storage.

[12] J. Zhang, A. Sivasubramaniam, H. Franke, N. Gautam,Y. Zhang, and S. Nagar. Synthesizing Representative I/O Workloads for TPC-H. In Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), 2004.

[13] S. W. Lee, D. J. Park, T. S. Chung, W. K. Choi, D. H. Lee, S.W. Park, and H. J. Song. "A log buffer based flash translation layer

using fully associative sector translation," ACM Transactions on Embedded Computing Systems, vol. 6, no. 3, 2007.

[14] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp.161–170, 2006.

[15] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems. SIGOPS Operating Systems Review, 42(6):36–42, 2008.

[16] Sang-Phil Lim, Sang-Won Lee, Bongki Moon, "FASTer FTL for Enterprise-Class Flash Memory SSDs" 2010 International Workshop on Storage Network Architecture and Parallel I/Os, pp.3-12, 3 May 2010

[17] X.-Y. Hu and R. Haas, "The fundamental limit of flash random write performance: Understanding, analysis and performance modelling," IBM Research Report, RZ 3771, Mar. 2010.

[18] http://memcached.org/

[19] http://aws.amazon.com/jp/elasticache/

[20] http://www.facebook.com/note.php?note_id=39391378919

[21] Molka, D.; Hackenberg, D.; Schone, R.; Muller, M.S.; , "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on , vol., no., pp.261-270, 12-16 Sept. 2009

[22] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J. Wright, 2010, "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud," In: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, p. 159–168, Washington, DC, USA.

[23]Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages

[24] http://hbase.apache.org/

[25] http://hbase.apache.org/book/datamodel.html

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears. “Benchmarking Cloud Serving Systems with YCSB,” Proc. SoCC '10, pp.143-154, Indianapolis, USA, Jun. 2010.

[27] Biplob Debnath, Sudipta Sengupta, Jin Li. “FlashStore: High Throughput Persistent Key-Value Store,” Proc. VLDB, vol.3, no.2, pp.1414-1425, Sep. 2010.

[28] Biplob Debnath, Sudipta Sengupta, Jin Li. "SkimpyStash: RAM Space Skimpy Key-Value Store on Flashbased Storage," Proc. SIGMOD '11 Conference, pp.25-36, Athens, Greece, Jun. 2011.

[29] http://aws.amazon.com/dynamodb/

[30] Intel X18-MX25-M SATA Solid-State Drive - 34 nm Product Line Product Specification, Order Number: 322296-007US, March 2011
http://download.intel.com/design/flash/nand/mainstream/Specification322296.pdf

[31]Product Specification Addendum for the Intel X18-MX25-M SATA Solid-State Drive - 34 nm Product Line, Order number 322697-002US, October 2010.
http://www.intel.com/design/flash/NAND/mainstream/pdf/322697.pdf

[32] B. H. Bloom. “Space/time trade-offs in hash coding with allowable errors.” Communications of the ACM, 13(7):422–426, 1970.

[33] Andrie Z. Broder and Michael Mitzenmacher. “Network Applications of Bloom Filters: A Survey,” Internet Mathematics, vol.1, no.4, pp.485-504, Jan. 2005.

[34] J. Haas and P. Vogt, “Fully-buffered dimm technology moves enterprise platforms to the next level” Technology@Intel Magazine, March 2005.

[35] Intel® Server Board S5520UR and S5520URT Technical Product Specification, Order number E44031-012,
http://www.intel.com/content/www/us/en/motherboards/server-motherboards/server-board-s5520ur.html

[36] Port Multipliers
http://www.sata-io.org/technology/port_multipliers.asp