Vol.2016-OS-138 No.9
2016/8/8

ブロックストレージシステムにおけるキャッシュの高速化

加藤 純[†,1]　佐藤 充[†]

概要：3D XPoint に代表される次世代メモリーの台頭により，DRAM キャッシュとストレージデバイスとの性能差が小さくなり，キャッシュ自体の性能がブロックストレージシステムの性能を決める重要な一指標となりつつある．Linux の Device Mapper フレームワークを使ったキャッシュのプロトタイプを評価したところ，4KiB Read/Write キャッシュヒットは理論限界性能 5499 K IOPS に対して 929/211 K IOPS と低く，IOPS が I/O 負荷にあわせてスケールしないことが分かった．本稿ではページ管理処理の遅延化と Linux の RCU (Read-Copy-Update)を用いた基数木によりロック競合を短縮することで高 IOPS を実現する手法を提案する．Device Mapper のオーバーヘッドを削減したことと合わせて，4KiB Read/Write キャッシュヒットは 3428/692 K IOPS と上記のプロトタイプと比較して 3.7/3.3 倍に向上した．

キーワード：ブロックストレージシステム，DRAM キャッシュ，Device Mapper，RCU，基数木

Cache Optimization in Block Storage System

JUN KATO[†,1]　MITSURU SATO[†]

Abstract: Future memory devices such as 3D XPoint have been closing the performance gap between DRAM cache and storage device and the storage cache itself can make a big impact on the overall performance in block storage system. Our prototype storage cache built on the Linux device mapper framework marks just 929/211 K IOPS in 4KiB Read/Write cache hit, while it can achieve 5499 K IOPS theoretically, and demonstrates a scalability limit. To break the limit for high IOPS, we propose cache optimization techniques, which reduce the lock contention by delaying page management and using RCU-protected radix tree. The techniques and reducing the overhead of the device mapper framework boost 4KiB Read/Write cache hit up to 3428/692 K IOPS, which is 3.7/3.3 times higher than the above prototype.

Keywords: Block Storage System, DRAM Cache, Device Mapper, RCU, Radix Tree

1. はじめに

3D XPoint[1]に代表される次世代メモリーの台頭により，DRAM キャッシュ自体の性能がブロックストレージシステムの性能を決める重要な指標として注目されつつある．ストレージデバイスが HDD から NAND 型フラッシュメモリー，次世代メモリーと高性能化していく中，ストレージシステムにもデバイスに合わせた性能が求められてきたが，DRAM キャッシュが行うソフトウェア処理よりも高速なハードウェアである次世代メモリーの登場[2]により，キャッシュ自体が性能低下の要因になりうる．容量単価の点から NAND 型フラッシュメモリーと次世代メモリーのハイブリッド型のストレージシステムが主流になっていくと予想されるが，性能的にこれら 2 つのデバイスの中間にあたるキャッシュはシステム全体の性能を引き出す際にボトルネックとなるコンポーネントであり，システム全体の性能を設計する上でその性能は重要な指標である．

本稿では，エンタープライズ向けの HA(High Availability)構成な共有ブロックストレージシステムを目的として実装したプロトタイプを用いて現状の課題の洗い出しを行う．Linux でブロックデバイスを扱うデファクトスタンダードの Device Mapper フレームワーク[3]を用いて実装・評価したところ，4KiB Read/Write キャッシュヒットの性能が理論限界性能 5499 K IOPS に対して 929/211 K IOPS と低いことが分かった．この主要因はページ管理を行うためのロックの競合である．キャッシュはページ単位で処理を行うためキャッシュ処理の開始時点でまずページの取得を試みるが，ページはシステム全体の共有リソースであるためロックにより保護されており，このシステムレベルのロック競合が IOPS の低下を引き起こす．

プロトタイプ評価の考察を踏まえて，システムレベルのロックの競合を減らして高 IOPS を実現する高速化手法を提案する．提案手法ではページ管理処理を I/O 完了後に遅延させることでロック期間を短縮し，競合を減少させる．また，ページの管理構造体に RCU(Read-Copy-Update)[4]保護の基数木を採用することによりページのヒット・ミス判定のためのロックを排除し，最終的にシステムレベルのロックは空きページの管理のみとした．実装上オーバーヘッドとなりうる Device Mapper フレームワークを不採用にして直接ブロックデバイスを扱うよう軽量化したことと合わせて，4KiB Read/Write キャッシュヒットは 3428/692 K IOPS とプロトタイプと比較して 3.7/3.3 倍に向上した．

† (株)富士通研究所
Fujitsu Laboratories LTD.
1 jun.kato@jp.fujitsu.com

2. 関連研究

キャッシュに関する研究のメインストリームはヒット率を向上させるためのページの置換アルゴリズムであり，LRU(Least Recently Used) [5]と LFU(Least Frequently Used)[6]アルゴリズムを基本として，ワンタイムなシーケンシャルアクセスからヒット率の高いページを保護する Scan-Resilience をはじめて導入した LRU-k[7]，複雑な LRU-k を FIFO と LRU のみで構成することで軽量化した 2Q[8], LRU と LFU を融合させたハイブリッドな LRFU[9]，IRR(Inter-Reference Recency)という新しい指標を導入した LIRS[10], L2 キャッシュ向けの MQ[11]，LRU の inclusion property[5]を活用するゴーストによりワークロードに合わせてキャッシュ内部の自動調整を行う ARC[12]などがある．

これらの置換アルゴリズムは 1 種類のページを管理するアルゴリズムであるが，ストレージシステムのキャッシュはストレージデバイスへの書き込みデータの有無によりページを Clean と Dirty の 2 種類に分類して管理するため，Clean 用と Dirty 用の管理にそれぞれ置換アルゴリズムが必要である．Clean 用と Dirty 用で同じアルゴリズムを用いることができるが，Dirty キャッシュは上書き（Dirty ページへのページヒット）によるストレージデバイスへの書き込み(Destage)回数の削減の他にデバイスの性能を最大限引き出せるようにページの書き込み順番・タイミングを決める役割があるため Clean 用とは異なるアルゴリズムになることがある．例えば，書き込み順番に関しては LRU ベースの時間的局所性に加えて空間的局所性として磁気ヘッドの移動距離も考慮した WOW[13]，WOW に Destage 速度も考慮した STOW[14], タイミングに関しては Dirty のページ数やその変化率から動的に決定する手法[15]などがある．また，Clean と Dirty の 2 種類に分割するためそれぞれのキャッシュサイズを決める必要があるが，ワークロードに合わせて動的にそれぞれのキャッシュサイズを決める手法としてゴーストを活用して Read ヒット率が最大になるように自動調整する AWOL[16], ARC[12]を階層的に構成することでゴーストをキャッシュサイズの自動調整にも転用する H-ARC[17]や I/O-Cache[18]などがある．

本稿は，キャッシュ自体の性能向上を目的としており，これまでのキャッシュによる性能向上を目的とした研究とは直交した研究である．これまでキャッシュはストレージデバイスに対して一桁以上速かったためキャッシュ自体の性能が遡上に上がることがなかった．実際，2Q[8]や ARC[12]に代表される LRU ベースの置換アルゴリズムはアルゴリズム上たかだか数回のリスト操作だけで実現されており非常に軽量である．そのような中，キャッシュ自体が本当にボトルネックになることがあるのか？次章では，プロトタイプの評価を通してこのことを定量的に確認する．

3. プロトタイプ評価

表 1: サーバーの緒元

機種	Fujitsu PRIMERGY RX200 S8
CPU	Intel Xeon E5-2697 v2
	2.70GHz, 24 コア(HTT 有効)
メモリー	DDR3 1600MHz, 128 GiB
IB HCA	Mellanox Connect X3, FDR 4X 56Gbps
OS	CentOS 7.1, kernel 3.10.0-229.11.1.el7

表 1 に示す緒元のサーバー2 台から構成される HA ペア上にストレージシステムのプロトタイプを実装して，評価を行った．プロトタイプではコンバージドプラットフォームとしてアプリケーションは FC や iSCSI などのストレージネットワークを経由せず，サーバー内のブロックデバイスに対して直接 I/O を発行する．これは，ストレージネットワークをバイパスすることでストレージシステム内部に閉じた性能を測るためである．ブロックデバイスに関しては，Linux でデファクトスタンダードである Device Mapper フレームワーク[3]を用いた．

プロトタイプではユーザーからの I/O は以下のように処理する．まず，I/O 範囲に関する排他処理を行い，I/O 範囲がオーバーラップする Write 処理をすでに実行中の場合は，先行する Write 処理が完了するまで待ち合わせる．先行する Write 処理がない場合，Read 処理は I/O 範囲がオーバーラップしていても並列に実行できる．この I/O 範囲に関する排他処理が取得できると，I/O を行うページ単位ごとにキャッシュのヒット・ミス判定を行う．ページは LUN (Logical Unit Number)番号と LBA (Logical Block Address)をキーとしてハッシュで管理を行う．ハッシュに該当ページが登録されていなかった場合がミスであり，ページの置換アルゴリズムを用いて空けたページを該当ページとしてハッシュに登録する．ページが取得できると，Read 処理の場合は必要なデータをストレージデバイスからページに読み出し(Stage), Write 処理の場合はページにデータをコピーして HA ペアに対してデータと LUN 番号や LBA などのメタデータのミラー処理を行う．HA ペア間は Infiniband で接続されており，ミラー処理は RDMA Write を用いて行う．

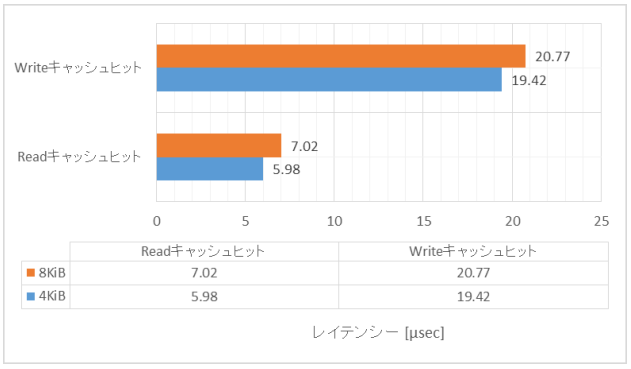

図 1: 4KiB Read/Write キャッシュヒットのレイテンシー

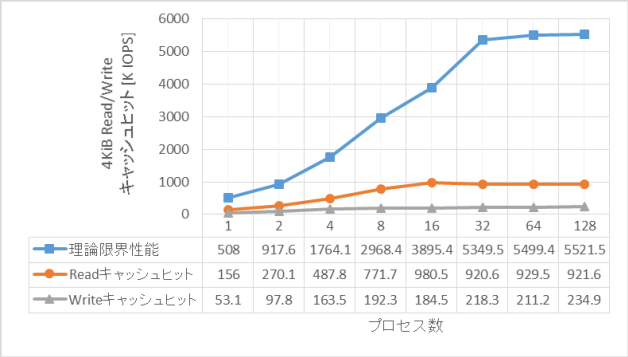

図 2: 4KiB Read/Write キャッシュヒットの IOPS

図 1，図 2 がそれぞれベンチマークツールである fio 2.2.8 を用いて測定した 4KiB Read/Write キャッシュヒットのレイテンシーと IOPS である．図 1 から Write キャッシュヒットは Read キャッシュヒットに比べて 3 倍近くレイテンシーが高いことが分かる．ページへのデータのコピーの向きを除けば Read と Write の処理の違いはミラー処理だけであり，この差が 13~14 マイクロ秒の差を生み出している．また，最速の 4KiB の Read キャッシュヒットでも 6 マイクロ秒近くと次世代メモリーのナノ秒オーダーに比べて一桁以上遅く，このことがキャッシュ，ひいてはソフトウェアがボトルネックになる可能性を示唆している．

4KiB Read/Write キャッシュヒットの IOPS は図 2 から理論限界性能 5499 K IOPS に対して 929/211 K IOPS と低く，I/O 負荷をかけるプロセス数を増やしても IOPS は 8~16 プロセス程度で頭打ちになる．本稿は，ブロックストレージシステムにおけるキャッシュをターゲットとしているが，ファイルシステムのキャッシュでも同様にキャッシュヒット時の性能がスケールしない課題が報告[19]されており，キャッシュヒットのスケーラビリティーはキャッシュの良し悪しを判断する重要な指標である．本研究の動機はこのスケーラビリティーボトルネックを改善することであり，ユーザー性能に直結するキャッシュの性能を改善することでユーザーに見えるシステム性能を改善することである．

4. ボトルネック箇所の検討

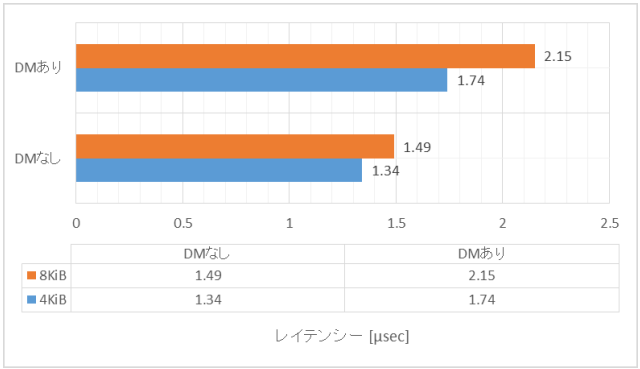

図 3: Device Mapper の有無によるレイテンシーの違い

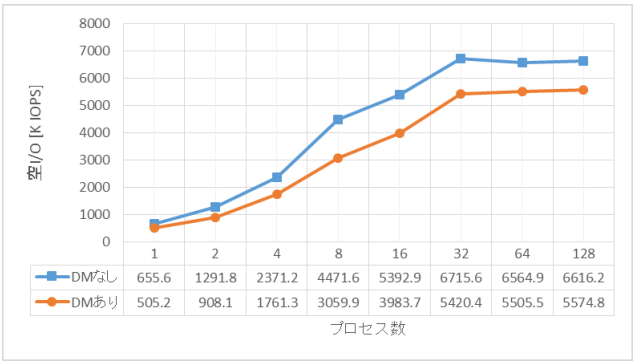

図 4: Device Mapper の有無による IOPS の違い

本章では，キャッシュヒットのスケーラビリティーを低下させるボトルネック箇所を検討する．まずは，ブラックボックスとして活用していた Device Mapper (DM)フレームワーク[3]のオーバーヘッドを検討する．図 3，図 4 は Device Mapper フレームワークを用いたブロックデバイスとプリミティブなブロックレイヤーAPI を用いたブロックデバイスのそれぞれで空 I/O を処理した際のレイテンシーと IOPS である．レイテンシーに関しては 4KiB で+0.4 マイクロ秒，8KiB で+0.66 マイクロ秒と Read/Write キャッシュヒット時のレイテンシーに比べて一桁以上小さく無視できるが，IOPS は-1000 K IOPS 近くと無視できない値である．Device Mapper フレームワークは Linux でブロックデバイスを扱う際のデファクトスタンダードであるため，今後コミュニティーによる次世代メモリーに対応した性能改善が期待できるが，現状は 15%の IOPS 低下が見込まれるため本稿のように性能を追求する場合は適切ではない．

次に，I/O フロー上のスケーラビリティーボトルネックを検討する．I/O フローの上でボトルネックとなる箇所は 2 箇所あり，ページを取得する際のシステムレベルのロックと Write 時のミラー処理である．前者について，ページはシステムレベルの共有リソースであるためヒット・ミス判定のためにページを管理するハッシュとページの置換アルゴリズムで使う LRU 等のリストの 2 つをシステムレベルのロックで保護しなければならない．各 I/O 処理はまずこのロックを取得しなければページを取得できないため後続の処理に進められないが，システムレベルのロックであるため競合しやすくスケーラビリティーのボトルネック要因となりうる．本稿では，このシステムレベルのロックが低スケーラビリティーを引き起こす主要因と捉え，次章でこの課題の解決に向けて取り組む．後者について，図 2 からミラー処理により Infiniband の帯域が 56Gbps 中 8Gbps 未満しか使われていないにも関わらず Write はプロセス数を増やしても 1 プロセス時の 4 倍程度と Read の 6 倍程度と比べて若干小さく，ミラー処理によりスケーラビリティーが低下している．このボトルネックに関しては，今後の課題として 6 章であらためて取り上げる．

5. 提案手法

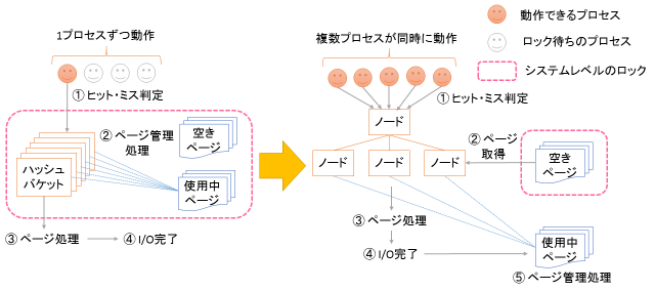

図 5: プロトタイプと提案手法での I/O フローの比較

システムレベルのロックによる保護期間の短縮に向け，提案手法では図 5 のようにページ管理処理を I/O 完了後に遅延させる．ページ管理はシステムレベルで行われるためその管理データ構造である LRU 等の操作にはシステムレベルの排他制御が不可欠であるが，新たにページ管理処理専用のスレッドを用意して，そのスレッドが I/O 処理が完了したページに対して一括してページ管理処理を行うように I/O フローを変更することで I/O 処理中にページ管理処理のためにシステムレベルのロックを取得する必要がなくなり，ロックの競合によるスケーラビリティーの低下を防ぐことができる．また，副次的な効果としてページ管理処理が I/O 処理中のレイテンシーに計上されなくなるので，ユーザーから見たレイテンシーの短縮も見込める．

I/O 処理中のシステムレベルのロックはページ管理処理のほかにキャッシュのヒット・ミス判定のためにも必要であったが，提案手法ではヒット・ミス判定のためのページ管理のデータ構造をハッシュから後述の基数木に変更して RCU(Read-Copy-Update)[4]により保護を行うことでヒット・ミス判定を複数プロセスが同時に行えるようにする．ページ管理処理の遅延化と合わせると，システムレベルのロックによる保護が必要となるのは空きページの管理だけとなり，前章で述べたスケーラビリティーのボトルネックであったシステムレベルのロック期間を大幅に短縮できる．

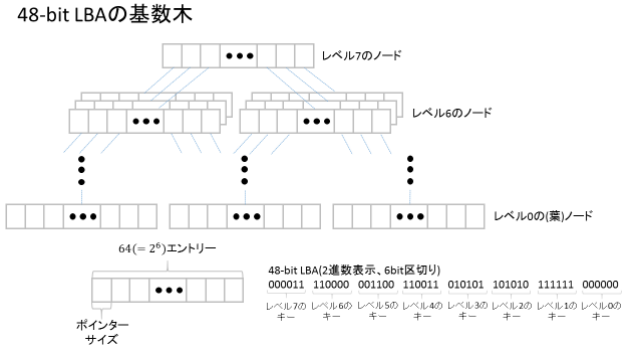

図 6: 48-bit LBA の基数木

基数木とは図 6 に示すように木を構成するノードにポインターサイズの複数のエントリー（図では 64 個）があり，キーの並びによって辿るノードを決める木構造である．以降，ノードを明示するために葉ノードをレベル 0 のノード，レベル 0 の親ノードをレベル 1 のノード，というようにレベルn(≥ 0)の親ノードをレベル n+1のノードと呼ぶことにする．提案手法では，LUN ごとに基数木があり，ページは葉ノードのエントリーに登録される．ヒット・ミス判定では，LBA をキーとして葉ノードの検索を行い，葉ノードの該当するエントリーにページがあればヒット，それ以外はミスとする．LBA 範囲が 48-bit，ノードのエントリー数が64(= 2^6)の場合，LBA を 6-bit ずつ 8 つに分割して最上位ビット側から順に高位レベルのキーとして用いる．例えば，LBA が 2 進数，6-bit ずつ/で区切った 000011 / 110000 / 001100 / 110011 / 010101 / 101010 / 11111 / 000000 の場合，レベル 7 のキーは 000011（10 進数で 3），レベル 6 のキーは 110000（10 進数で 48）である．キャッシュのヒット・ミス判定のために葉ノードの検索を行う場合，レベルnのノードではレベルnのキーをインデックスとしてエントリーをひくことで子ノードを辿っていく．上述の LBA の例では，レベル 7 のキーは 3 のため 3 番目のエントリーを，レベル 6 のキーは 48 のため 48 番目のエントリーを，と順々にノードを辿っていく．葉ノードのキーに当たるエントリーには子ノードの代わりにページが入り，キャッシュヒットの場合にはこのようにしてページを見つける．

RCU(Read-Copy-Update)[4]はリーダー・ライターロックに似たリソースの保護機構であり，リーダーのオーバーヘッドが通常のリーダー・ライターロックに比べて極めて小さいことが特徴である．キャッシュのヒット・ミス判定はページの参照カウント管理と合わることでリーダーの保護範囲内で処理できるので，基数木を RCU で保護することにより，複数プロセスでのヒット・ミス判定を同時にかつ軽量に行うことができる．実際，Linux のファイルシステムキャッシュではページ管理に RCU 保護の基数木が使われており[20]，次章の評価では Linux の RCU 保護の基数木をベースにブロックストレージシステム向けの対応，後述するライターによる保護範囲の局所化とコンパクション処理を実装した RCU 保護の基数木を用いる．

ミスをしてノード割り当てを行う際のライターの保護範囲を基数木単位からノード単位に局所化することでミス時の処理，特に Write ミスのスケーラビリティーを向上させる．Read はミスをした場合 Stage が発生するためストレージデバイスが性能を律速させるが，Write ではこのノード割り当て処理と空きページの取得があることがヒットとミスの違いであり，空きページが不足して Destage による処理が発生しない限り，スケーラビリティーを低下させるのはノード割り当て処理だけである．ライターの保護範囲をノード単位に局所化することで，1 つの LUN に対してバースト的に Write ミスが発生したとしてもライターの保護範囲がノードに分散されているため競合が緩和され，スケーラビリティーを向上させることができる．

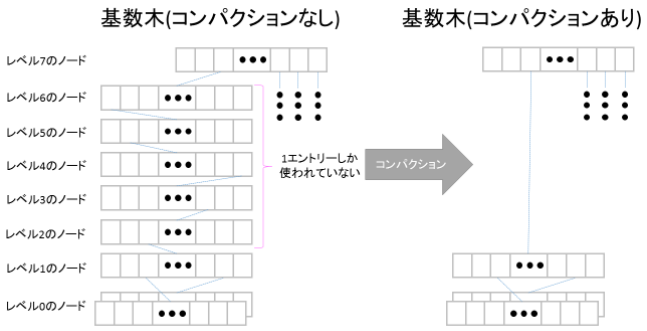

図 7: 基数木のコンパクション

図 6 のような素朴な基数木ではランダムアクセスのような空間的局所性の低いワークロードの際に空間オーバーヘッドが大きくなるため，図 7 に示すコンパクションを行いノード数の削減をすることでオーバーヘッドを軽減する．基数木の各ノードは子ノードやページを指すためのポインターサイズのエントリーを複数（典型的には 64）持つため，空間的局所性が低いと 1 ノードあたりの使われないエントリー数が増えて空間オーバーヘッドが大きくなる．そこで，レベル 1 以上の葉ではないノードにおいて 1 エントリーしか使われていないノードの割り当てを行わず，2 エントリー以上が使われるようになるまでノードの割り当て処理を遅延させる．その場合，エントリーは子孫に当たるノードを直接指すが，子ノードを省略したためヒット・ミス判定の際に本来子孫ではないノードが見つかることがあり，素朴な基数木の検索のままでは誤判定を引き起こす．そこで，子ノードを辿る際はノードの担当範囲から正しい子孫であるか確認を行い，誤った子孫であればその時点でミスと判定する処理を新たに加える．

6. 評価と今後の課題

提案手法をプロトタイプの評価で用いた表 1 の緒元のサーバーから構成される HA ペア上に実装して評価を行った．プロトタイプではブロックデバイスの実装に Device Mapper フレームワークを用いたが，4 章の検討から Device Mapper フレームワークが 1000K 近くの IOPS 低下を引き起こすことが分かったので，評価に当たってプリミティブなブロックレイヤーの API を用いて書き直した．

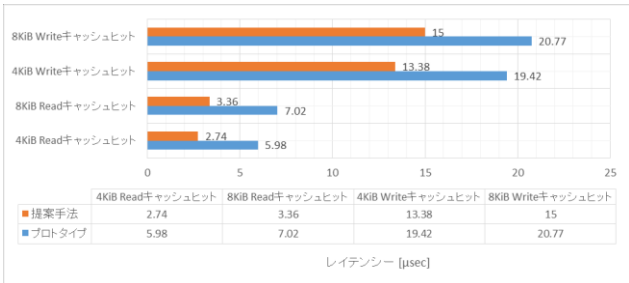

図 8: プロトタイプと提案手法でのレイテンシー比較

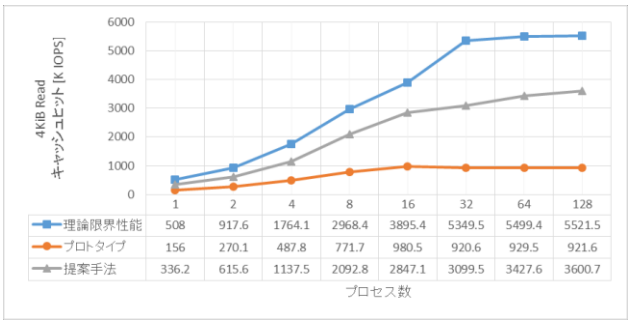

図 9: プロトタイプと提案手法での
4KiB Read キャッシュヒットの IOPS 比較

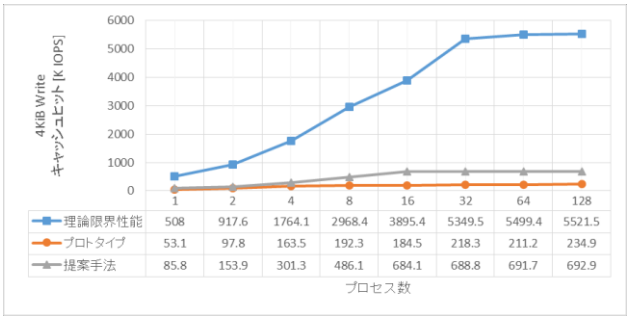

図 10: プロトタイプと提案手法での
4KiB Write キャッシュヒットの IOPS 比較

図 8，図 9，図 10 はプロトタイプと提案手法でのレイテンシーと 4KiB Read/Write キャッシュヒットの IOPS を比較した図であるが，提案手法を用いることで課題であったスケーラビリティーが改善され，4KiB Read/Write キャッシュヒットは 3428/692 K IOPS とプロトタイプの 929/211 K IOPS と比較して 3.7/3.3 倍に向上した．図 8 のレイテンシーに注目すると Read で 4 マイクロ秒，Write で 6 マイクロ秒短縮した．これは Device Mapper フレームワークのオーバーヘッドを削減した実装上の効果と，ページ管理処理を I/O 完了後にまで遅延化させたことによりそのオーバーヘッドがレイテンシーに計上されなくなったためである．図 9 の Read キャッシュヒットの IOPS ではプロトタイプが 8~16 プロセス程度で IOPS が頭打ちになっているのに対して，提案手法を用いることで 128 プロセスまで IOPS が負荷をかけるプロセス数に比例してスケールしている．表 1 の緒元から CPU コア数が 24 のため 32 プロセスを超えるとスケーラビリティーは鈍化するが，CPU コア数が十分に足りている間は理論限界性能に近いスケーラビリティーを示している．このことから，4 章で検討したシステムレベルのロックが IOPS 低下を引き起こす要因であったことが，逆説的ではあるが確認でき，また提案手法によりシステムレベルのロック競合が緩和されスケーラビリティーが改善したことも分かる．Write キャッシュヒットの場合も図 10 から同様にスケーラビリティーが改善されるが，16 プロセス程度で頭打ちになり，IOPS の絶対値も Read キャッシュヒットの 20%程度である．

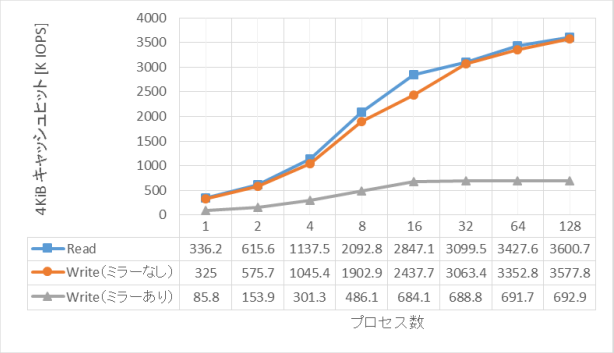

図 11: Read と Write(ミラーあり，なし)の IOPS 比較

Write キャッシュヒットの IOPS が Read キャッシュヒットの 20%程度と低くなるのはミラー処理があるからである．図 11 は提案手法での Read とミラーあり，なしの Write の IOPS を比較した図であるが，ミラーを行うことで Write の IOPS が 3352 K から 691 K へと Read キャッシュヒットと比較した時と同じく 20%程度に低下している．これは，提案手法によりシステムレベルのロック競合が緩和されたことで，4 章であげたもう 1 つのボトルネック要因であるミラー処理が新たなボトルネックとして浮かび上がったためである．このことから，今後のスケーラビリティー改善の鍵を握るのはミラー処理であり，その解決への糸口に向けて以下ではミラー処理について考察を行う．

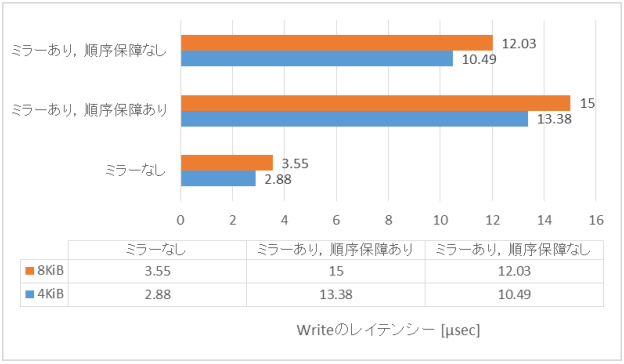

図 12: Write 処理のレイテンシー比較

ミラー処理では RDMA Write による通信オーバーヘッドに加えて HA を実現するためのデータとメタデータの通信順番もオーバーヘッドとなる．そのことを示したのが図 12 である．Write のミラー処理ではミラー先のデータの整合性を保つためデータのミラーが完了してからメタデータのミラー処理を行う．この処理を直列化してスケーラビリティーを低下させる順序保障のオーバーヘッドを評価するため，試しに順序保障を無視してデータとメタデータを並列にミラーしたところ，レイテンシーは 3 マイクロ秒程度短縮された．データとメタデータの RDMA Write による通信オーバーヘッドはミラーなしの場合と比較して 8 マイクロ秒程度であることから，順序保障によるオーバーヘッドは 27%程度と高く，この順序保障の制約をうまく緩和することでスケーラビリティーが改善する可能性がある．

ミラー処理が律速する要因は帯域ではなく，通信データサイズが小さいために Infiniband のコントローラーがボトルネックになるためである．図 10 から提案手法を用いても Infiniband の帯域は 22Gbps 程度しか使われておらず，表 1 の緒元の 56Gbps と比較しても帯域は半分も使われていない．キャッシュは I/O を数 KiB のページ単位に小さく分割して処理を行うこと，加えてデータの他にページ単位で数十 B の小さなメタデータもミラーするため帯域を使い切るよりも先にコントローラーがボトルネックになる．そのため，スケーラビリティーの改善に向けてコントローラーの負荷が下がるようにミラー処理の通信パターンを変えることが重要であり，どのように行うかは今後の課題である．

7. おわりに

キャッシュはブロックストレージシステムの性能を決める重要なコンポーネントであるが，従来ではキャッシュヒットの IOPS が I/O 負荷にあわせてスケールしない．このことを Device Mapper フレームワークベースのプロトタイプを通して実証し，システムレベルのロックの競合とミラー処理がボトルネックであることを確認した．本稿では，前者のボトルネックを，ページ管理処理を遅延させる I/O フロー上の，また RCU 保護の基数木を採用するデータ構造上の 2 つの新方式を適用することでシステムレベルのロック競合を緩和する手法を提案した．また，実装上 Device Mapper フレームワークがオーバーヘッドとなりうることを示し，そのオーバーヘッドを削減した実装上の改善も含めて 4KiB Read/Write キャッシュヒットで 3428/692 K IOPS とプロトタイプと比較して 3.7/3.3 倍の IOPS を実現した．

今後の課題として，Write キャッシュヒットの IOPS 改善がある．Write キャッシュヒットの IOPS は Read キャッシュヒットの 20%程度であり，これは本稿では扱わなかったもう 1 つのボトルネックであるミラー処理が IOPS を低下させるためである．ミラー処理がボトルネックとなる要因は，データとメタデータの通信に順序保障が必要なこと，通信データサイズが小さく帯域を使い切る前にコントローラーがボトルネックになること，の 2 つである．今後は Write キャッシュヒットの IOPS 改善に向けて，この 2 つのボトルネック解消に取り組む予定である．

謝辞 実装と評価に当たり，富士通 ストレージシステム事業本部のみなさまから多数のアドバイス・コメント・フィードバックをいただき大変お世話になりました．ここに深く感謝の意を表します．また，富士通の塩沢 賢輔氏，富士通研究所の前田 宗則氏と松尾 勇気氏にはアイデアレビューから実装・評価のサポートまで，本研究を進めるにあたって多大なご協力およびご支援をいただきました．心から感謝を申し上げます．

参考文献

[1] “3D XPoint”. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html, (参照 2016-07-13).
[2] S. Swanson and A.M. Caulfield. Refactor, Reduce, Recycle: Restructing the I/O Stack for the Future of Storage. Computer, 2013, vol. 46, no. 8, p. 52-59
[3] “Device Mapper”. http://www.sourceware.org/dm/. (参照 2016-07-13)
[4] D. Guniguntala et al. The Read-Copy-Update Mechanism for Supporting Real-Time Applications on Shared-Memory Multiprocessor Systems with Linux. IBM Systems Journal, 2008, vol. 47, no. 2, p. 221-236
[5] R.L. Mattson et al. Evaluation Techniques for Storage Hierarchies. IBM Systems Journal, 1970, vol. 9, no. 2, p. 78-117
[6] A.V. Aho et al. Principles of Optimal Page Replacement. Journal of the ACM, 1971, vol. 18, no. 1, p. 80-93
[7] E.J. O'neil et al. The LRU-k Replacement Algorithm for Database Disk Buffering. ACM SIGMOD 1993, vol. 22, no. 2, p. 297-306
[8] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. VLDB, 1994, p. 439-450
[9] D. Lee et al. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Transactions on Computers, 2001, vol. 50, no. 12, p. 1352-1361
[10] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. ACM SIGMETRICS Performance Evaluation Review, 2002, vol. 30, no. 1, p. 31-42
[11] Y. Zhou et al. Second-Level Buffer Cache Management. IEEE Transactions on Parallel and Distributed Systems, 2004, vol. 15, no.6, p. 505-519
[12] N. Megiddo and D.S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. USENIX File and Storage Technologies, 2003, vol. 3, p. 115-130
[13] B.S. Gill and D.S. Modha. WOW: Wise Ordering for Writes – Combining Spatial and Temporal Locality in Non-Volatile Caches. USENIX File and Storage Technologies, 2005, vol. 4, p. 129-142
[14] B.S. Gill et al. STOW: A Spatially and Temporally Optimized Write Caching Algorithm. USENIX Annual Technical Conference, 2009, p. 29-42
[15] S. Faibish et al. A New Approach to File System Cache Writeback of Application Data. SYSTOR, 2010
[16] A. Batsakis et al. AWOL: An Adaptive Write Optimizations Layer. USENIX File and Storage Technologies, 2008, p. 1-14
[17] Z. Fan et al. H-ARC: A Non-Volatile Memory Based Cache Policy for Solid State Drives. 30th Symposium on Massive Storage Systems and Technologies, 2014, p. 1-11
[18] Z. Fan et al. I/O-Cache: A Non-Volatile Memory Based Buffer Cache Policy to Improve Storage Performance. 23rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2015, p. 102-111
[19] C. Min et al. Understanding Manycore Scalability of File Systems. USENIX Annual Technical Conference, 2016, p. 71-85
[20] S. Rao et al. Examining Linux 2.6 Page-Cache Performance. Proceedings of the Linux Symposium, 2005, vol. 2, p. 79-90