Vol.2016-OS-138 No.15
2016/8/9

KVSを利用した高速なブロックストレージ

田所 秀和[1]　長谷川 揚平[1]　石山 政浩[2]　松崎 秀則[1]

概要：スケーラビリティの向上を目的として Key Value Storage (KVS) を利用したストレージシステムが使われつつある。このようなストレージシステムでは、KVS を複数利用して *Storage Pool* を作成し、ユーザは Object や Block など用途に合わせた *Storage Interface* を通して Storage Pool 上の KVS を利用する。今般、SSD などの高速なストレージデバイスを活用して KVS を高速化する技術が進展しており、それと並行して Storage Interface の高速化も重要になりつつある。そこで本稿では、高速な KVS やネットワーク環境を十分に利用することができる高速な Block Interface を提案する。このシステムでは Linux® カーネルの Multi-Queue を利用しつつ、効率良く KVS と通信を行うことで、高い性能を達成している。memcached を用いた実験により、4KB ランダムリードで 1Miops を越える性能を確認した。

Hidekazu Tadokoro[1]　Yohei Hasegawa[1]　Masahiro Ishiyama[2]　Hidenori Matsuzaki[1]

1. はじめに

Ceph [1] や vSAN [2]、RAMCloud [3] のように、Key Value Storage (KVS) を利用したストレージシステムが使われつつある。KVS とは、GET や PUT のような単純な API を持つストレージコンポーネントである。Key と呼ばれる文字列のみでデータにアクセスできるシンプルなデータモデルを採用している。そのため、KVS を利用することで領域を柔軟に管理することができる [4]。このようなストレージジステムでは、複数の KVS をネットワークに接続し、ソフトウェアを用いて束ねることにより、スケーラブルな *Storage Pool* を作成できる。Storage Pool の容量を増やすには、必要に応じて KVS をネットワークに追加するだけでよく、管理コストを下げることができる。ユーザは、このような Storage Pool に対して利用したい *Storage Interface* を通してアクセスする。オブジェクトストレージとしてアクセスする場合には Object Interface、VFS としてアクセスする場合には VFS Interface を利用する。OS 領域として利用するなど、既存のファイルシステムを利用したい場合には、Block Interface を通してファイルシステムを作りそれをマウントして使う。Block Interface は、既存のファイルシステムなどのソフトウェア資産を生かすために重要なソフトウェアである。図 1 は、CephFS における Storage Pool と Storage Interface の様子である。

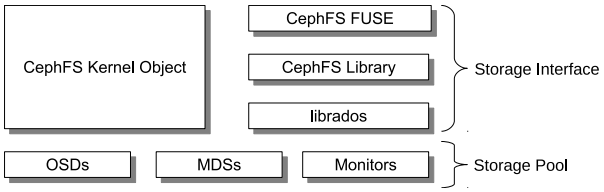

図 1　Ceph での Storage Pool と Storage Interface ([5] より作成)

既存の技術によって、高速な Storage Pool を構築することができる。KVS に関連する技術では、Flash に最適化された高速な KVS が数多く研究されている [6] [7] [8] [9] [10] [11] [12] [13]。SSD は HDD と比較してランダムアクセス性能で優れている。また、内部並列性が高くソフトウェアの工夫によりより高スループットなシステムを構築することができる [14]。また、10Gbps/40Gbps イーサネットのような高速なネットワーク技術もデータセンターなどで一般的になりつつある [15]。高速な KVS を高速なネットワーク繋ぐことで、大容量で高性能な Storage Pool を作ることが可能になる。

高速な Storage Pool の技術と比較して、既存の Block Interface は遅いという問題がある。例えば Ceph では、個々の OSD ではなく Object Interface を提供する RADOS 上に仮想ディスクをファイルを置き、Block Interface を実装している。信頼性は RADOS が担保してくれるため実装が簡単になるが、一方で Block Interface を実現するための階層が増えてしまいランダム性能を犠牲にしてしまう。実際、予備実験により Ceph block interface の性能を測定した

[1] (株) 東芝 研究開発センター
[2] (株) 東芝 ストレージ&デバイスソリューション社

ところ、主記憶上に OSD を構築を利用した環境でも、約 50Kiops 程度のランダムリード性能であった。DRAM と比較して著しく性能が低く、Ceph Block がデバイスの性能を出し切れていないことがわかる。

Block Interface を利用する既存のアプリケーションは、数百 us 程度のレイテンシや数 Gbps の帯域といったローカルストレージデバイスを想定して作られている。そのため、性能の低い Block Interface を使うことで、アプリケーション自体の性能が低下してしまう。このように、従来の KVS を利用したストレージでは、Block Interface の性能は重視されていなかった。

本稿では、KVS を利用した高速なブロックストレージを提案する。このシステムは、KVS によって作られた Storage Pool に対して、高速で柔軟に拡張可能な Block Interface を提供する。提案システムは並列性を生かして、個別の KVS にダイレクトにアクセスすることにより高い性能を発揮することができ、高速な KVS やネットワーク環境を十分に利用することができる。また、既存のライブラリを利用して拡張可能であることが特徴である。現状の KVS に使われるプロトコルは、memcached や Redis、kinetic など複数存在し、場合によって使い分けることが多い。そのため、既存ライブラリを利用して新しい KVS への対応を実装できる利点は大きい。また、例えば重複排除のような新しい機能を、既存の高速なライブラリを使って実現することができる。

実験により、提案システムの性能を測定し、4KB ランダムリード性能で 1Miops の性能を確認した。また、ランダムリード以外の場合では、KVS かファイルシステムかネットワークがボトルネックになり、提案システムは現状の環境では十分な性能があることを確認できた。

以下、2 章では KVS ストレージの問題点について述べ、3 章で高速なブロックストレージを提案する。4 章では提案システムを用いた実験について述べる。5 章で関連研究に触れ、6 章でまとめる。

2. ストレージシステムにおけるブロックストレージ

大量のデータを扱うコンピュータシステムでは、KVS が重要な役割を担っている。KVS は、GET や PUT、Delete のような単純な API と、Key と Value というシンプルなデータモデルを持つストレージコンポーネントである。memcached [16] に代表されるように、DRAM を使い高速に実装されていることが多い。例えば、重複排除 [11] [7] [12] や画像のキャッシュ [17]、Web インデクシング [18] など、大量のデータを高速に処理する現代のコンピュータシステムでは必須の要素となっている。

KVS は、システムの高速化のためだけではなく、ストレージシステムの構成要素としても使われている。このようなシステムでは Storage Pool と呼ばれるデータを保存する階層と、論理的なビューを提供する階層である Storage Interface が存在する。KVS をベースとするストレージシステムの利点は、3 つある。1 つ目は、既存の IP network 設備を流用することで、システムを安価に構築できることである。現在のデータセンターなどは IP を基本として通信システムを構築しているため、これらの設備をそのまま利用することが可能である。2 つ目は、KVS 由来のスケーラビリティのおかげで、容易に容量を増やすことが可能である。データモデルと API を工夫することで、レイテンシと一貫性を両立するスケーラブルな KVS を実現することも可能である [3]。3 つ目は、新しい機能を追加しやすいことである。Storage Interface と Storage Pool を分けることにより、新しいプロトコルへの対応は、新しい Storage Interface を開発するだけでよい。データの保存などは Storage Pool にまかせればよいため、新しい機能の開発だけに注力できる。Storage Interface には、VFS Interface [5, 19] や Object Interface [20]、Block Interface [21] が存在する。

Storage Interface の中でも、Block Interface は依然として重要である。Block Interface とは、0 から始まる連続アドレスによってセクターサイズの領域を指定する方法である。ファイルシステムは信頼性が重要であるため、ユーザは xfs や ext4 など実績のあるファイルシステムを使う傾向にある。これらの実績のあるファイルシステムは、Block Interface を用いてデータを読み書きする必要がある。そのため、ストレージシステムにおいて Block Interface の提供が重要である。また、Block Interface 上のファイルシステムには、OS など性能が重視されるファイルが置かれることが多い。大量のデータは Object Interface を通して、効率的に保存されることが多く、Block Interface は容量よりもランダム IO 性能が重視される。

Storage Pool の性能を向上する技術が提案されている。多くの、Flash や SSD に最適化された KVS [6–13] が提案されてきている。これらの技術を使うことで Flash や SSD の性能を引き出し、より高速な Storage Pool を作ることができる。また、10/40GbE といった高速なネットワーク技術が一般に使われるようになっており [15]、高速な KVS を組み合わせて、より高速な Storage Pool を作成することが可能である。

このようにバックエンドである Storage Pool の性能を向上させる技術はあるが、KVS をバックエンドとする Block Interface の性能は、我々の知る限り、あまり向上していない。実際に Ceph Block の性能を測定する実験を行った。主記憶上にディスクを作成し、クライアントと 40GbE で接続することで、できる限りフロントエンドの性能を測定できるようにした。図 2 がその結果である。ランダムリード性能が約 50Kiops、ランダムライト性能が約 17Kiops と、DRAM の性能よりも著しく低い値となった。これは、仮

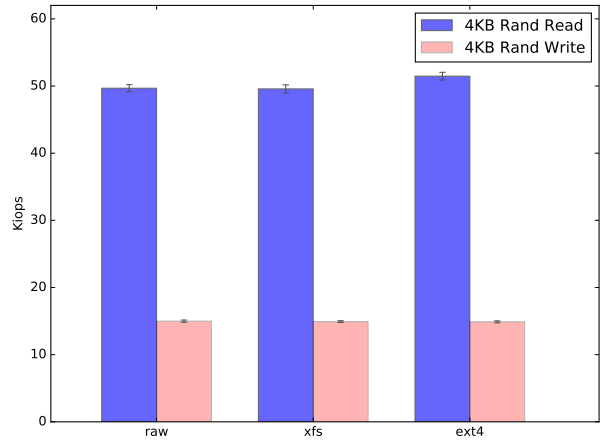

図 2　Ceph Block のランダム IO 性能

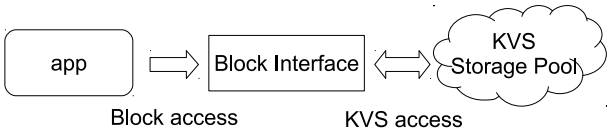

図 3　提案システムの概要

想ディスクの実装がオブジェクトストレージ上にファイルとして作られているため、ランダムアクセスに弱いからである。

3. KVS を利用した高速なブロックストレージ

2 章で述べた問題を解決するため、KVS をバックエンドストレージとして利用する高速なブロックデバイスを提案する。このシステムは、ブロックアクセスと KVS アクセスとのプロトコル変換を高速に実行することができる。図 3 に示すように、アプリケーションに対しては Block Interface を提供し、実際のデータは KVS を利用して保存する。

3.1 概要

ホスト側は Linux3.18 以降を対象とし、KVS として memcache に対応している。構成は図 4 に示すように、大きく分けてカーネルモジュールである *KVBmodule* とユーザランドプロセスである *KVClient* から成る。KVBmodule は、ブロックデバイスとキャラクタデバイスを作成し、アプリケーションと KVClient の間で IO リクエストをやりとりする。ブロックデバイスは、ファイルシステムなど通常のファイルシステムを利用するために使われる。キャラクタデバイスは、KVClient とのやりとりのために使われる。KVClient は、IO リクエストと KVS リクエストの変換と KVS との通信を担当する。起動時に KVS との TCP コネクションを確立しておき、Read や Write など IO の種類に応じて KVS とやりとりをする。IO 処理の前半では、キャラクタデバイスから IO リクエストを受けとり、KVS リクエストに変換した後にソケット経由で KVS にリクエストを出す。IO 処理の後半では、KVS からレスポンスを読み取り、IO レスポンスに変換してからキャラクタデバイスに書き込む。

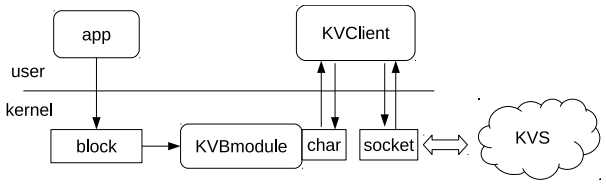

図 4　カーネルモジュールとユーザランドのハイブリッド実装

3.2 特徴

提案システムの特徴は次の 2 つである。

3.2.1 KVS 処理のユーザランド実装

KVS へのアクセス部分をユーザランドで実装している。ユーザランドで実装している利点として、既存のライブラリを利用しやすいことが挙げられる。KVS のプロトコルは、memcached や Redis、kinetic など複数存在し、場合によって使い分けることが多い。そのため、既存ライブラリを利用して新しい KVS への対応を実装できる利点は大きい。もし、すべてカーネルで実装した場合、ライブラリをカーネルへ移植したり、新たに実装し直すなどの手間がかかってしまう。

また、KVS プロトコル実装部を KVClient としてユーザプロセスに分離することで安全に拡張することができる。KVClient は、外部からのアクセスを解釈する役割を担うため、外部からのアクセスに晒されやすい。また、プロトコルの実装は、バッファ管理など複雑になりやすくバグを埋め込みやすい。その結果、攻撃の起点になりやすいため、より安全な実装が求められる。ユーザランドで実装した場合には、ある程度信頼できる既存のライブラリも使うことが可能である。

一方で、このような実装では、ユーザランドとカーネル間でのデータコピーが多発し、性能上の問題がある。1 回 IO を処理するために、KVClient は 4 回カーネルとやりとりする必要がある。(i) IO リクエストデータの受け取り、(ii) KVS へのリクエスト送信、(iii) KVS からのレスポンス受信、(iv) IO レスポンスの送信の 4 つである。このような問題に対して、KVClient は効率よく IO を発行することで性能を向上させている。複数の IO をまとめて単一のシステムコールで処理することで、1 回あたりのシステムコールのコストを削減している。詳しい実装については 3.4 章で詳しく述べる。

3.2.2 高い並列性

各階層の並列性をうまく生かすことにより、マルチコア環境を生かせる設計になっている。ブロックレイヤでは Linux カーネルの Multi-Queue [22] を利用し、各コアで独立に処理を可能にした。図 5 に示すように、ユーザランド部もマルチスレッドで動作させ、コアごとに存在する

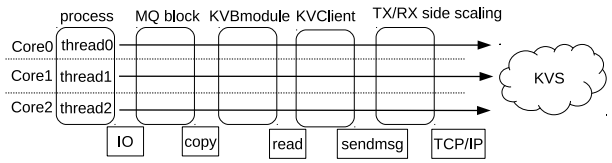

図 5　コアごとに独立性が高い実装

Multi-Queue と 1 対で対応させた。これにより、CPU コア間で通信が発生せずに並列に処理することができる。さらに、ユーザランド部では各コアごとに KVS に対して TCP コネクションを張り、独立にネットワーク IO を実行する。TCP コネクションを複数利用することにより、Receive Side Scaling を効率よく働かすことができる。

3.3 KVS の利用方法

提案システムでは、LBA をキーとして 4KB バイトのデータを KVS に保存することで、Block Interface を実現している。IO リクエストのサイズが大きい場合、KVClient は分割して KVS への複数リクエストに変換する。このとき、KVS からの複数のレスポンスから、IO レスポンスを構成する必要がある。そのため、KVClient は IO リクエストをどのように分割したかの状態を覚えている。SSD は内部並列性が高いため [14]、SSD に最適化された KVS を使うこと想定した場合、このように IO リクエストを分割して KVS に処理させる方式は性能向上に有利だと考えられる。

3.4 イベント駆動アーキテクチャ

ユーザとカーネルを分離したアーキテクチャになっている。詳細な IO の手順は次の通りである。

1. アプリケーションが IO リクエストを発行する
2. ブロックデバイスが IO リクエストを受け取り、KVB-module へ渡す
3. KVBmodule は受け取った IO リクエストをキャラクタデバイスを利用して KVClient へ渡す
4. KVClient は受け取った IO リクエストを対応する KVS リクエストに変換する
5. その KVS リクエストをソケット経由で KVS に投げる
6. KVS から応答が返ってきたら、KVClient はその KVS レスポンスを IO レスポンスに変換する
7. KVClient が IO レスポンスをキャラクタデバイスに書き込む
8. KVBmodule はアプリケーションに対して完了を通知する

KVClient はキャラクタデバイスとソケットそれぞれのリード・ライトを効率良く実行する必要がある。これらの IO はシステムコールで実装されているため遅く、大量に発行すれば性能に悪影響を与える。性能を向上させるため、KVClient はイベント駆動とバッファリングを活用して、効率よく IO を発行している。イベント駆動プログラミングを利用することで、IO の待ち時間を浪費せずにサービスの実行を継続できる。イベント駆動プログラミングを実現するために、キャラクタデバイス側で epoll システムコールを実装した。KVBmodule は、IO リクエストが到着するとキャラクタデバイスが読み込み可能になったと epoll 経由で KVClient に通知する。イベント駆動プログラミングで実装された KVClient は、epoll 経由で通知を受けると、自分のタイミングでキャラクタデバイスをリードする。キャラクタデバイスの書き込みは、IO を完了するためだけであり、常に書き込み可能状態を維持している。ソケット IO のイベント駆動については、Linux で既に実装されているものを利用した。

KVClient によるキャラクタデバイスとソケットの読み書きは、バッファリングで高速化している。複数の IO リクエストや KV リクエストを、バッファリングによって少ない回数のリード・ライトで取得する。これにより、IO リクエストごとにシステムコールを発行するよりも回数を削減でき、性能向上に繋がる。KVBmodule は複数の IO リクエストを単一のリード・ライトで受け渡しできるよう、キャラクタデバイスを実装している。十分なバッファを指定してリードシステムコールを発行すれば、KVBmodule に現在到着している全ての IO リクエストを、1 回で取得することが可能である。KVClient は、これらイベント駆動プログラミングとバッファリングを、libevent [23] を利用して実装した。

4. 実験

提案システムの性能を評価する。実験環境は、

- CPU: Intel® Xeon® E7-4890 v2 2.80GHz 60core
- Memory: 512GB
- NIC: Mellanox MT27500 ConnectX-3 40GbE
- Linux 4.5

KVS 側は、

- CPU: Intel® Xeon® E5-2687W v3 3.1GHz 20core
- Memory: 256GB
- NIC: Mellanox MT27500 ConnectX-3 40GbE
- Linux 4.4

を用い、これらのマシンを 40GbE Switch を介してで繋いだ。KVS として memcached-1.4.25 [16] を利用した。

4.1 KVS の性能

3.3 章で述べたように、提案システムは IO リクエストを 4K バイト単位に区切って KVS に保存するため、4K バイト単位での KVS の性能が提案システムの性能に大きく影響する。提案システムへの影響を調べるため、4K バイトでの memcached の性能を測定した。ベンチマークツールには mutilate [24] を用い、さまざまな負荷でのスループッ

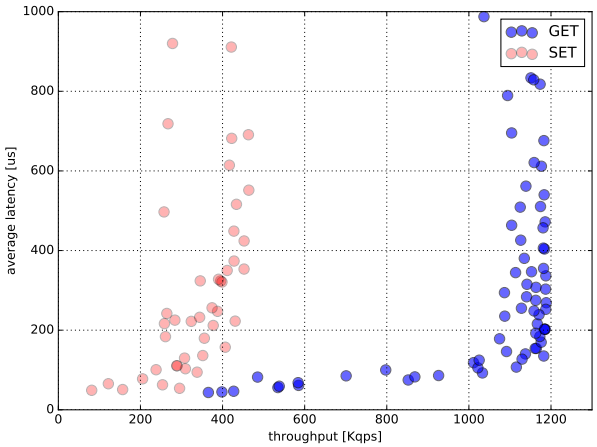

図 6　KVS として使用した memcached の性能

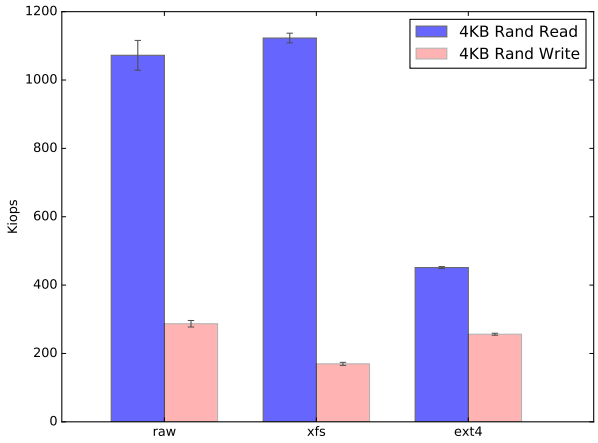

図 7　ランダム IO 性能

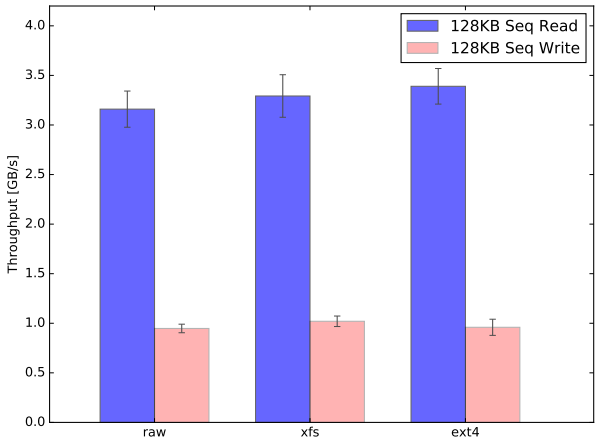

図 8　シーケンシャル IO 性能

トとレイテンシを測定した。

図 6 が結果である。全体の傾向として負荷をかけていくとスループットが上昇し、ある水準を越えると急激にレイテンシが上昇する。この急激にレイテンシが上昇するポイントが限界性能だと考えられる。GET では 1.2Mqps の性能が出ている。ほぼ 40Gbps のネットワークを使い切っており、ネットワークがボトルネックになっている。また、SET では 400Kqps 程度の性能であった。memcached は SET が遅い傾向にある。詳しく調べていないが、内部での実装に用いているハッシュテーブルのロックが原因ではないかと推測している。この結果から、4KB ランダムリードの場合 1.2Miops、ランダムライトの場合、400Kiops が限界性能であると考えられる。

4.2　提案システムの性能

提案システムの性能を評価した。性能は、ランダム IO、シーケンシャル IO、レイテンシを測定した。ファイルシステムの影響を調べるため、ブロックデバイスから直接測定した場合と、ファイルシステムを介した場合の性能を測定した。ファイルシステムは xfs と ext4 を用いた。KVClient はユーザスレッド数 20、KVS との通信では TCP コネクション数 20 で測定した。ベンチマークソフトとして fio [25] を用いた。ランダム IO ではジョブ数 20、IO Depth32 で測定した。レイテンシではジョブ数 1、IO Depth1 で測定した。

図 7 がランダム IO 性能である。4KB ランダムリードにおいて、raw と xfs では 1Miops を越える性能を確認できた。ほぼ 40Gbps の性能を使い切っており、ネットワーク帯域がボトルネックになっている。一方で ext4 の性能が 400Kiops 程度と悪い。原因は不明であるが、ext4 の実装のまずさがランダムリード性能に影響を与えている可能性がある。4KB ランダムライトでは、200Kiops から 300Kiops 程度の性能であった。KVClient の問題のためか、memcached の限界性能である 400Kiops には届かなかった。また、xfs での性能が他に比べて低く、ファイルシステムの性能も影響している可能性がある。

次に、シーケンシャル IO 性能を調べた。図 8 が 128KB シーケンシャル IO 性能の結果である。シーケンシャルリード性能が、シーケンシャルライト性能を大きく上まわっている。これは、KVS として利用する memcached が、GET のほうが SET より性能が高いためである。シーケンシャルリード性能は 3.2GB/s 程度であった。ext4 を除く 4KB ランダムリードと比較して、スループットが出ていないことがわかる。これは、提案システムが IO を分解して KVS へリクエストを出すため、複数の KVS リクエストを管理するためにオーバーヘッドがかかるためだと考えられる。一方で、シーケンシャルライトは約 1GB/s であり、ランダムライトとスループットで大きな差はなかった。また、ファイルシステムによる違いは見られなかった。

次に、1IO を処理するのにかかる時間を調べた。図 9 が結果である。60us から 70us の時間がかかっていることがわかる。ping での round trip time は 70us 程度であり、ネットワークレイテンシが支配的であることがわかる。ファイルシステムやリード・ライトによるレイテンシの違いは見られなかった。

これらの結果から、いずれの場合も KVS やネットワー

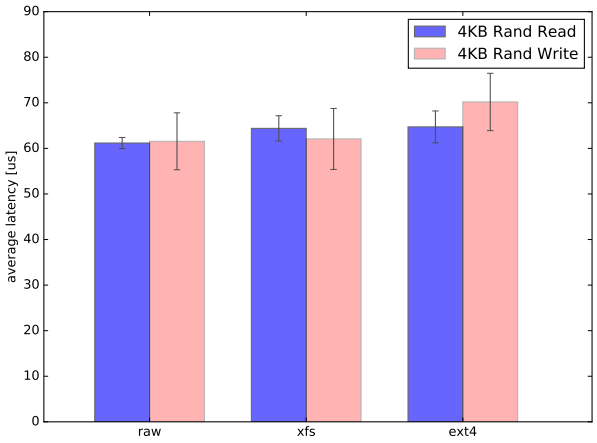

図 9　レイテンシ性能

ク・ファイルシステムがボトルネックになっていることがわかった。提案システムが現状の環境と比較して十分高速であると言える。

5. 関連研究

Tyche [26, 27] は、Ethernet を経由した高速なブロックストレージである。TCP/IP のオーバーヘッドを避けるために、Ethernet 上に独自プロトコルを実装している。DRAM を利用した実験では、4KB ランダム IO で約 300Kiops 程度、シーケンシャル IO で約 50Gbps 程度である。我々の提案システムでは、KVS/TCP という汎用プロトコルを利用しているところが異なる一方で、同じような性能を達成できている。

Ceph Block [21] は、Ceph における Block Interface である。カーネルモジュールを通して提供され、実データは OSD 上に保存される。RADOS プロトコルの実装もカーネルモジュールとして実装されるため、カーネルモードで動くコードベースが大きくなる。

iSCSI [28] や ATA over Ethernet, Fiber Channel over Ethernet, nbd [29]、HyperSCSI [30] は、ネットワーク越しにブロックデバイスを提供する。これらは、ネットワーク上にブロック用のプロトコルを実装しており、KVS のような汎用プロトコルを利用した提案システムとは異なるものである。

Blizzard [31] は、クラウドストレージ上に Block Interface を作るシステムである。クライアント側のドライバがブロックアクセスを受け取り、クラウド側のストレージにデータを保存する。ストレージとして、FlatDataStorage というものを想定しており、一般的な KVS とは異なる。また、クラウド環境をターゲットとしているため、レイテンシが数 ms から数十 ms のオーダーを想定している。我々のシステムでは、より高速な環境を想定している。

Salus [32] は、HBase 上に Block Interface を作るシステムである。一貫性の維持に重点を置いており、一部のストレージが壊れても、prefix semantics によりある時点での一貫した状態に戻ることができる。また、クラウド環境をターゲットとしているため、レイテンシが数 ms から数十 ms のオーダーを想定している。

RAMCloud [3] は、DRAM を用いた高速で低レイテンシなストレージシステムを提案している。Key-Value をベースとしてスケーラビリティを実現し、DRAM をメインに使うことで高い性能を実現している。RAMCloud では、アプリケーションが KVS の API を叩くことを前提としており、Block Interface は提供していない。RAMCloud と提案システムとを組み合わせることで、高速なブロックストレージシステムを構築可能である。

6. まとめと今後の課題

本稿では、KVS を利用した高速なブロックストレージを提案した。提案システムでは、拡張性が高く高速なアーキテクチャを採用している。ユーザカーネル分割アーキテクチャによって安全に拡張可能で、CPU コア数に対してスケールするアーキテクチャによって高速に動作する。実験により、4KB ランダムリード性能で 1Miops を越えていることを確認した。また、多くの場合、KVS かファイルシステムかネットワークがボトルネックになり、現状の環境では十分な性能があることを確認できた。実験に利用した memcached は SET 性能が低く、より高速な KVS の作成が必要である。ファイルシステムのなど既存のソフトウェアレイヤの抜本的な改良も必要だと考えられる。raw device では十分なランダムリード性能があるにもかかわらず、ファイルシステムを経由すると遅い場合があり、高速なストレージシステムを実現するには高速なソフトウェアレイヤの研究が重要である。

参考文献

[1] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn: Ceph: A Scalable, High-Performance Distributed File System, *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 307–320 (2006).

[2] VMware, Inc.: Virtual SAN and Object-Based Storage, `http://pubs.vmware.com/vsphere-55/index.jsp#com.vmware.vsphere.storage.doc/GUID-2B3B720F-0A7E-4B4B-883F-85A39C1A6C5A.html`.

[3] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang: The RAMCloud Storage System, *ACM Transaction on Computer Systems*, Vol. 33, No. 3, pp. 7:1–7:55 (2015).

[4] Erik Riedel and Sami Iren: Object Storage and Applications, *Proceedings of the 2007 Linux Storage & Filesystem Workshop*, LSF '07.

[5] Inktank Storage, Inc.: Ceph Filesystem, `http://docs.ceph.com/docs/master/cephfs/`.

[6] Hyeontaek Lim, Bin Fan, David G. Andersen and Michael Kaminsky: SILT: A Memory-efficient, High-performance Key-value Store, *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 1–13 (2011).

[7] Biplob Debnath, Sudipta Sengupta and Jin Li: FlashStore: High Throughput Persistent Key-value Store, *Proc. VLDB Endow.*, Vol. 3, No. 1-2, pp. 1414–1425 (2010).

[8] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan and Vijay Vasudevan: FAWN: A Fast Array of Wimpy Nodes, *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 1–14 (2009).

[9] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala and Raju Rangaswami: NVMKV: A Scalable, Lightweight, FTL-aware Key-value Store, *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pp. 207–219 (2015).

[10] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau: WiscKey: Separating Keys from Values in SSD-Conscious Storage, *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pp. 133–148 (2016).

[11] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath: Cheap and Large CAMs for High-performance Data-intensive Networked Systems, *Proceedings of the 7th Symposium on Networked Systems Design and Implementation*, NSDI '10.

[12] Biplob Debnath, Sudipta Sengupta, and Jin Li: SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage, *In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11.

[13] Vijayendra Shamanna (Viju): Optimizing Ceph for All-Flash Architectures, *In Proceedings of the Vault Linux Storage and Filesystems Conference 2015*, Vault'15.

[14] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho: F2FS: A New File System for Flash Storage, *Proceedings of the 13th USENIX Symposium on File and Storage Technologies*, FAST '15.

[15] John D'Ambrosia: The Evolution of Ethernet, *Proceedings of the 26th Large Installation System Administration Conference*, LISA '12.

[16] Brad Fitzpatrick, et al.: memcached, `https://memcached.org/`.

[17] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel: Finding a needle in Haystack: Facebook' s photo storage, *In Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, OSDI '10.

[18] 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, *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06.

[19] Feng Wang, Scott A. Brandt, Ethan L. Miller and Darrell D. E. Long: OBFS: A File System for Object-based Storage Devices, *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '04.

[20] Inktank Storage, Inc.: Ceph Object Gateway, `http://docs.ceph.com/docs/master/radosgw/`.

[21] Inktank Storage, Inc.: Ceph Block Device, `http://docs.ceph.com/docs/master/rbd/rbd/`.

[22] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet: Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems, *In Proceedings of the 6th International Systems and Storage Conference*, SYSTOR'13.

[23] Niels Provos, et al.: libevent - an event notification library, `http://libevent.org/`.

[24] Jacob Leverich: mutilate, `https://github.com/leverich/mutilate`.

[25] Jens Axboe: fio, `https://github.com/axboe/fio`.

[26] Pilar González-Férez, and Angelos Bilas: Tyche: An efficient Ethernet-based protocol for converged networked storage, *In Proceedings of the 30th Sympoium on Mass Storage Systems and Technologies*, MSST'14.

[27] Pilar González-Férez, and Angelos Bilas: Reducing CPU and network overhead for small I/O requests in network storage protocols over raw Ethernet, *In Proceedings of the 31st Sympoium on Mass Storage Systems and Technologies*, MSST'15.

[28] J. Stran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner: RFC3720: Internet Small Computer Systems Interface (iSCSI), `https://www.ietf.org/rfc/rfc3720.txt`.

[29] Pavel Machek: Network Block Device, `https://atrey.karlin.mff.cuni.cz/~pavel/nbd/nbd.html`.

[30] Wilson Yong Hong Wang, Heng Ngi Yeo, Yao Long Zhu, and Tow Chong Chong: Design and development of Ethernet-based storage area network protocol, *In Proceedings of the 12th IEEE International Conference on Networks*, ICON '04.

[31] James Mickens, Edmund B. Nightingale, Jeremy Elson, Krishna Nareddy, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan: Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications, *In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14.

[32] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin: Robustness in the Salus Scalable Block Store , *In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13.