IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 14 July 2023, accepted 21 July 2023, date of publication 24 July 2023, date of current version 31 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3298552

Holistic In-Network Acceleration for Heavy-Tailed Storage Workloads

GYUYEONG KIM, (Member, IEEE)
Department of Computer Engineering, Sungshin Women's University, Seoul 02844, Republic of Korea
e-mail: gykim@sungshin.ac.kr

This work was supported by the Sungshin Women's University Research Grant of 2022.

ABSTRACT Storage workloads are typically heavy-tailed, and a small number of large requests incur a burdensome performance overhead. To this end, we present NetStore, an in-network storage accelerator that exploits the capability of emerging programmable switches. The key idea of NetStore is to directly process large requests in the network by leveraging switches as an in-network request processor. NetStore not only mitigates head-of-line blocking but also provides extra computational power for data storage. To overcome the strict resource constraints of the switch ASIC, we take a holistic approach that carefully co-designs the switch data plane and the switch control plane. Specifically, we design a custom control plane that acts as a dedicated request processor and a custom data plane that performs size-aware request scheduling and large object tracking. Our solution can be implemented on a commodity programmable switch at a line rate using only 6.82% of the switch memory. We implement a NetStore prototype on an Intel Tofino switch and conduct a series of testbed experiments. Our experimental results demonstrate that NetStore can improve throughput, the median latency, and the 99th percentile latency by up to 1.19×, 21.29×, and 2.91×, respectively.

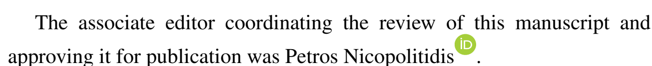
INDEX TERMS Programmable data plane, P4, in-network computing.

I. INTRODUCTION

Today's online services like web search and social networking require a number of data read to process user requests, thereby highly relying on data storage, which is supported by in-memory key-value stores like Redis [1]. For high availability, object data is typically replicated over multiple servers (typically 3-5 [2]) using replication protocols [3], [4]. To meet strict user-facing Service Level Objectives (SLOs) [5], distributed storage should provide high throughput and low latency [6], [7]. Therefore, recent replication protocols like Hermes [2] and CRAQ [8] allow local reads in any storage replica for scalable read performance.

Meanwhile, storage workloads are typically heavy-tailed in object sizes [9], [10], [11], [12]. Most of the objects are tens to hundreds of bytes, and only a few objects are tens to thousands of kilobytes [9], [12]. In addition, the portion of the requests for large objects[1] is less than 1% [9]. Although the portion of large requests is small, their impact on the performance is substantial. Specifically, large requests consume many computing resources because they consist of multiple packets. Furthermore, they degrade the tail latency of small requests by causing head-of-line blocking in the storage server. Unfortunately, the existing replication protocols [2], [8] neglect the performance overhead of large requests by implicitly assuming that every object is small. While there are CPU scheduling solutions, they only mitigate head-of-line blocking and still cannot reduce the resource usage of large requests since they are intra-server solutions [9], [13].

In this paper, we ask the following question: *how can we avoid performance overhead caused by large requests to build high-performance replicated storage?* As the answer to the question, we present NetStore, an in-network storage accelerator that exploits the capability of emerging


The associate editor coordinating the review of this manuscript and approving it for publication was Petros Nicopolitidis.

[1]We use the terms 'requests for small/large objects' and 'small/large requests' interchangeably.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

programmable networking hardware. The key idea of NetStore is to move large request processing into the network by utilizing the high-performance programmable switch as an in-network large request processor. This in-network acceleration offers two fundamental advantages. First, since large requests are isolated, small requests are not blocked by large requests in the storage server, reducing the latency of small requests. Second, the switch provides extra computational power without adding a new server, thereby improving throughput. Our idea is based on the following observations: 1) the switch is a centralized place that can observe every message being exchanged between storage servers; 2) programmable switch ASICs like Intel Tofino [14] provide flexibility and capability to parse and process application messages by custom processing logic while supporting a Tbps-scale throughput.

Although we have an opportunity for in-network storage acceleration, it is hard to realize the idea due to strict resource constraints of switch hardware. Commodity programmable switch ASICs provide only a 10-20 MB on-chip memory and a limited number of computational match-action stages, making it hard to store and process large requests fully in the switch data plane. To overcome this, we take a holistic approach that exploits the synergy between the switch data plane and the switch control plane. The control plane has underutilized hardware components like a x86 processor and tens of GB memory. Considering the small portion of large requests, the switch control plane is an attractive place to process large requests without extra hardware costs. Therefore, we expose the switch control plane as a dedicated large request processor and make the switch data plane schedule requests by dynamically tracking large objects. This is a distinct feature that distinguishes NetStore from existing in-network solutions that only utilize the switch data plane [6], [7], [15].

To partition request handling functions between the data plane and the control plane carefully, we design custom data and control plane modules. The switch data plane modules include the size-aware read scheduling module and the large object tracking module. The size-aware read scheduling module forwards read requests to storage servers or the switch control plane according to the requested object size. To distinguish large requests, the large object tracking module maintains the list of large object IDs by coordinating the switch and storage servers. The module inserts the large object threshold into the custom header, and the storage server sets the object identification flag in the header after identifying large objects by comparing the threshold and the object size. The module inserts object IDs into the large object list only if the write is committed to avoid reading the stale data. Meanwhile, the request processing module in the switch control plane maintains large objects and serves large requests, which are forwarded by the switch data plane.

We implement a NetStore prototype on a commodity programmable switch with an Intel Tofino switch ASIC [14] using P4 language [16]. Since we store not the object data but the object IDs in the switch data plane, NetStore only uses 6.82% of the switch memory. To evaluate NetStore, we build a testbed consisting of 8 commodity servers connected to a 6.5 Tbps Intel Tofino switch and conduct a series of extensive experiments. Our key findings from experimental results include that: 1) compared to Hermes [2], the state-of-the-art replication protocol, NetStore improves throughput, the median latency, and the 99th percentile latency by up to 1.19×, 21.29×, and 2.92×, respectively; 2) NetStore can recover throughput rapidly in the presence of switch failures; 3) NetStore is robust to various system conditions like the portion of large requests, access patterns, and hash table sizes.

In summary, this paper makes the following contributions.

- We present NetStore, an in-network storage accelerator that supports high throughput and low latency for heavy-tailed workloads in replicated storage by leveraging programmable switches as an in-network request processor.
- We fully leverage the architectural opportunity of switch hardware by taking a holistic approach that co-designs the switch control plane and the switch data plane carefully.
- We implement a NetStore prototype on a commodity switch and conduct extensive experiments to demonstrate that NetStore provides better performance and is robust to switch failures and various system conditions.

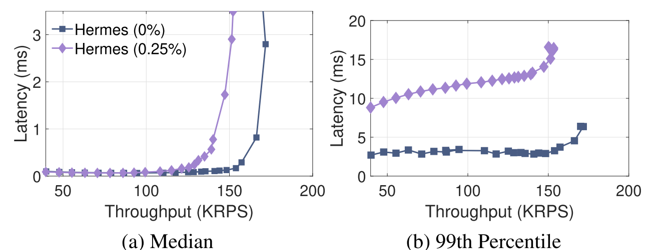


FIGURE 1. Impact of large requests on throughput and latency.

The remainder of the paper is organized as follows. In Section II, we describe the motivation of this work. Section III and Section IV provides the design rationale and design of NetStore, respectively. We present implementation and evaluation results in Section V and Section VI, respectively. We discuss related work in Section VII. Lastly, we conclude our work in Section VIII.

II. BACKGROUND AND MOTIVATION

A. HEAVY-TAILED OBJECT SIZE DISTRIBUTIONS

Modern online services depend on highly-available replicated data storage, which is supported by in-memory key-value stores like Redis [1]. A typical workload characteristic is that the object size distribution is heavy-tailed [10], [12], [17]. Most of the objects are small and only a few objects are large. For the Facebook workload [12], 55% of the objects are less than 128 bytes but 10% are between 1024 bytes to 1 MB. Similarly, in the Wikipedia workload [18], most of the objects are small, but 1% of the objects span from

4096 bytes to 1 MB. The portion of the requests for large objects is less than 1% [9].

B. PERFORMANCE OVERHEAD CAUSED BY LARGE REQUESTS

Although only a few objects are large, large requests can have a significant impact on storage performance, as described below. First, large requests require more CPU resources than small requests. This is because the server has to process multiple packets to handle a single large request, as the requested object size typically exceeds the maximum transmission unit (MTU) size of network interfaces. Second, large requests degrade the tail latency of small requests by causing head-of-line blocking, in which small requests wait excessively while large requests are processed by CPUs.

To motivate our work, we perform testbed experiments. The testbed setup is described in Section VI in detail. We use four clients and four storage replicas using Hermes [2], the state-of-the-art replication protocol. We use a heavy-tailed workload that has a bimodal distribution consisting of 128-B and 256-KB objects. The objects represent byte-scale small and KB-scale large objects [9], respectively. We use 40Gbps for the line speed of NICs and the performance bottleneck is CPUs. Figure 1 (a) and Figure 1 (b) show the median and tail read latency as throughput grows with and without large requests, respectively. Hermes (0.25%) means that 99.75% and 0.25% of the requests are for 128-B objects and 256-KB objects, respectively. We can clearly see that the throughput and latency are significantly degraded although the portion of large requests is only 0.25%. This is because, as we have aforementioned, the small requests are stuck by the large requests in the servers.

For replicated storage, we need a cluster-level solution that mitigates head-of-line blocking and reduces resource usage for large requests at the same time. However, existing works do not achieve the requirements. Recent replication protocols [2], [8] significantly improve the performance of replicated storage by allowing local reads, but they neglect the performance overhead of large requests in heavy-tailed workloads. Meanwhile, there exist multicore CPU scheduling solutions. For example, Shinjuku [13] uses preemption among requests and Minos [9]) isolates large requests in dedicated cores. These solutions are effective for head-of-line blocking in a single server, but they do not reduce the resource usage of large requests since the server still processes large requests.

III. A CASE FOR IN-NETWORK STORAGE ACCELERATION

A. DESIGN GOAL AND KEY IDEA

Our goal is to build replicated storage that supports high throughput and low latency simultaneously by minimizing the performance overhead of large request processing. To achieve the goal, we process large requests in the network instead of storage servers by leveraging switches as an in-network request processor. The switch is a central place that has a clairvoyant view of every message being exchanged between interconnected storage servers. This implies that the switch has an inherited chance to process large requests with a global view. Furthermore, emerging programmable switches provide enough flexibility and capability to handle large requests with custom processing logic.

Figure 2 illustrates the programmable switch architecture. For the switch data plane, programmable switch ASICs like Intel Tofino [14] provide a Tbps-scale processing throughput while allowing us to program the packet processing pipeline, which includes the packet parser, the Match-Action (M-A) stages, and the packet deparser. By customizing the parser and the M-A stages, the switch can identify custom storage messages and perform custom packet processing. Along with the switch data plane, the programmable switch contains the switch control plane consisting of an x86 CPU, DRAM, and SSD, which can perform general-purpose computing.

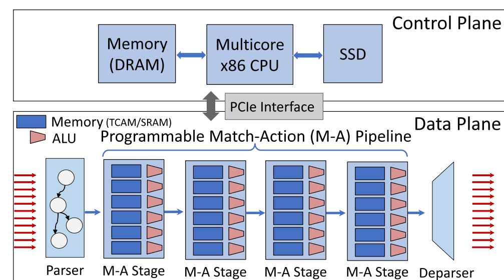


FIGURE 2. Programmable switch architecture. Both the switch control plane and the switch data plane provide opportunities to accelerate heavy-tailed storage workloads.

B. HOLISTIC IN-NETWORK ACCELERATION

Despite the flexibility, commodity programmable switch ASICs have fundamental resource constraints and timing requirements to support a Tbps-scale packet processing throughput. The switch ASICs provide only 10-20 megabytes of on-chip memory and a limited number of M-A stages for custom data processing. In each M-A stage, a small amount of memory and ALUs are statically allocated [19]. This indicates that it is hard to handle large requests fully in the switch data plane because the packet processing pipeline cannot handle data larger than $k \times n$ bytes where k is the supported number of M-A stages and n is the maximum data size that can be processed by ALUs per stage. For example, suppose that a switch ASIC is with $k = 8$ and $n = 16$. This means that the packet processing pipeline can handle data up to 128 bytes, which are not enough to store and read large objects.[2]

To overcome the constraints, we leverage the capability of the switch control plane as well, not only relying on the switch data plane. Specifically, we expose the switch control plane as a dedicated request processor that serves large requests. Meanwhile, the switch data plane acts as a request scheduler that tracks large object IDs and forwards

[2]The specific numbers for k and n depend on the switch architecture and cannot be publicly available due to the Non-Disclosure Agreement (NDA).

read requests to the switch control plane if the requested object is large. Our holistic approach is based on the following observations.

- The switch control plane is generally underutilized. The primary job of the switch control plane is to update entries of the local packet forwarding table. However, the forwarding table of the switch in distributed storage is rarely updated because storage clusters have a transparent and static network topology.
- The switch control plane is increasingly equipped with high-performance hardware components. For example, the APS Networks APS7232Q switch offers options for the switch control plane that includes an Intel 8-core Xeon CPU@2.7 Ghz and 64 GB DRAM. The Edgecore CSP-7550 is even with high-performance server-grade components that include two Intel 16-core Xeon Gold 5218 CPU@2.3 Ghz and 256 GB DRAM.
- The portion of large objects in heavy-tailed workloads is small, and the portion of large requests is also small. Therefore, although the control plane hardware components may have relatively lower computational power than the high-performance storage server, the switch control plane is suitable to serve large requests.

C. WHY NOT A SERVER-BASED SOLUTION?

We may implement the idea of NetStore using a storage server instead of programmable switches. In this design, a dedicated storage server processes large requests, and each client schedules requests in a distributed manner. However, this approach has two limitations as follows.

- Large objects are partially tracked by each client. Due to the partial information, a client forwards large requests to storage servers, though the requested object is already known as large by another client. Contrastively, NetStore uses the switch as a centralized scheduler, hence every client sees the global tracking information for large objects.
- The server-based solution wastes computing resources since the dedicated server does not process small requests that take most of the requests (>99%). The server consumes switch resources as well by occupying a switch port. However, our switch-based solution can naturally augment the system performance without adding a new server and every storage server can process any type of request.

D. CHALLENGES

While conceptually simple, it is not straightforward to translate the idea into a working system because of various technical challenges. The addressed challenges are as follows.

- What is the storage system architecture to realize the idea of in-network large request processing?
- How does the switch process requests efficiently based on the object size?
- How does the switch data plane track large object IDs?
- How can we handle various types of failures?

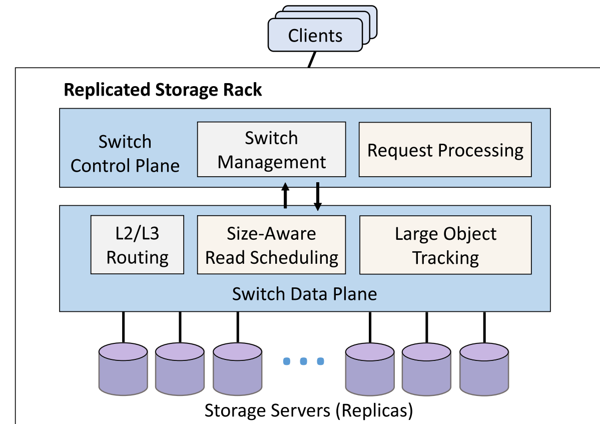


FIGURE 3. Replicated storage rack with NetStore.

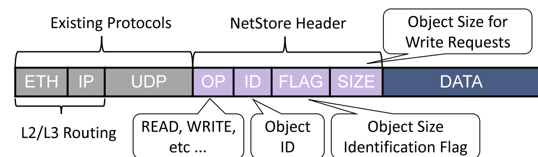


FIGURE 4. NetStore packet format.

IV. NetStore DESIGN

A. NetStore ARCHITECTURE

The NetStore architecture consists of the switch control plane, the switch data plane, storage servers, and clients. We illustrate the architecture in Figure 3.

1) SWITCH CONTROL PLANE

The switch control plane acts as the in-network large request processor that contains two modules, which are the switch management module and the request processing module. The switch management module establishes a communication channel between the control plane and the data plane. In addition, the module inserts table entries of M-A tables and system configuration values that are stored in registers (e.g., the large object threshold). The request processing module handles read and write requests for large objects by maintaining large objects in memory.

2) SWITCH DATA PLANE

The switch data plane acts as the request scheduler that consists of three modules as follows. The L2/L3 routing module performs traditional packet forwarding functions through port lookup for both NetStore packets and normal packets. The size-aware read scheduling module determines the destination of read requests depending on the size of the requested object. In particular, while small requests are forwarded to a storage server determined by the replication protocol, large requests are delivered to the switch control plane for in-network processing. The large object tracking module maintains the list of large object IDs by coordinating the switch data plane and storage servers. The module removes the object ID from the list (if exists) when handling write

Algorithm 1 Packet Processing in Data Plane

```
   – pkt: Packet to be processed
   – Obj: List of large objects
   – Srv: List of storage servers
   – Th: Large object threshold
1: if pkt.op == READ-REQUEST then
2:     pkt.flag ← Th                          ▷ Insert threshold
3:     if pkt.id ∈ Obj then                   ▷ Read for large object
4:         pkt.dst ← Control Plane            ▷ Update destination
5:     end if
6: else if pkt.op == WRITE-REQUEST then
7:     pkt.flag ← Th                          ▷ Insert threshold
8:     if pkt.id ∈ Obj then                   ▷ To prevent consistency issue
9:         Obj.Remove(pkt.id)                 ▷ Remove object ID
10:    end if
11: else if pkt.op == READ-REPLY or WRITE-REPLY then
12:    if pkt.flag == True then               ▷ Large object
13:        Obj.Insert(pkt.id)                 ▷ Add object ID to list
14:    end if
15: end if
16: Forward(pkt)
```

requests and inserts the ID again when committing the write. This prevents the client from reading outdated data in the switch control plane when pending writes for the requested object exist.

3) STORAGE SERVERS AND CLIENTS

NetStore requires only marginal modifications on servers and clients. Since NetStore does not get involved in write coordination and load balancing of read requests, the data storage can use existing replication protocols like Hermes [2] and CRAQ [8]. Clients basically send requests and receive replies. The only thing that the client does for NetStore is to insert the metadata (e.g., the operation type and the object ID) into the NetStore header between the UDP header and the data payload. Storage servers perform GET and PUT operations and stamp the object size identification flag to distinguish large objects. In addition, storage servers replicate only large objects in the switch control plane when handling write requests.

B. PACKET FORMAT

Figure 4 shows the packet format of NetStore. NetStore uses a custom L7 protocol message. We reserve a UDP port number for NetStore so that the switch can apply different packet processing logic for NetStore packets and normal packets. Normal packets are forwarded to a matched output port based on standard L2/L3 routing without being processed through NetStore stages, which means that NetStore is compatible with existing functions. The NetStore message has a header that consists of four fields as follows.

noitemsep

Algorithm 2 Object Size Identification in Storage Server

```
1: if pkt.op == READ-REQUEST then
2:     data ← GET(pkt.id)                     ▷ Read data
3:     if Len(data) > pkt.flag then           ▷ Is object large?
4:         pkt.flag ← True                    ▷ Yes, large
5:     else
6:         pkt.flag ← False                   ▷ No, small
7:     end if
8: else if pkt.op == WRITE-REQUEST then
9:     if pkt.size > pkt.flag then            ▷ Is object large?
10:        pkt.flag ← True                    ▷ Yes, large
11:    else
12:        pkt.flag ← False                   ▷ No, small
13:    end if
14: end if
```

- `OP`: the message operation type, which can be `READ-REQUEST`, `WRITE-REQUEST`, `READ-REPLY`, `WRITE-REPLY`. We also use additional types like `WRITE-UPDATE`, `WRITE-ACK`, which are used to update storage servers and the switch control plane during write coordination of replication protocols.
- `ID`: the ID of a requested object.
- `FLAG`: the identification flag for the object size. *True* means the requested object is large, whereas *False* indicates it is a small object. We also use this field to carry the large object threshold from the switch to the storage server.
- `SIZE`: the size of the requested object. Since read requests cannot know the object size in advance, this field is used only for write requests.

C. REQUEST PROCESSING

NetStore has a different processing logic depending on the type of message as follows. Algorithm 1 describes the pseudocode of request processing in the switch data plane.

1) READ REQUESTS

Upon receiving a read request, the switch data plane first inserts the large object threshold into the flag field to deliver the threshold to storage servers (line 2). Next, the switch data plane checks whether the requested object is included in the large object list (line 3). If the object is in the list, the switch data plane forwards the request to the switch control plane for further handling (line 4). Otherwise, the request is forwarded to a storage server. Since NetStore is not a load balancing solution, we rely on replication protocols to determine the initial destination server. We currently use Hermes [2] that selects a random replica at the client for both read and write requests. The switch data plane finishes read processing by forwarding the packet (line 16).

When the switch control plane receives the read request, the switch control plane simply returns the requested object data to the client. The storage server works similarly to the

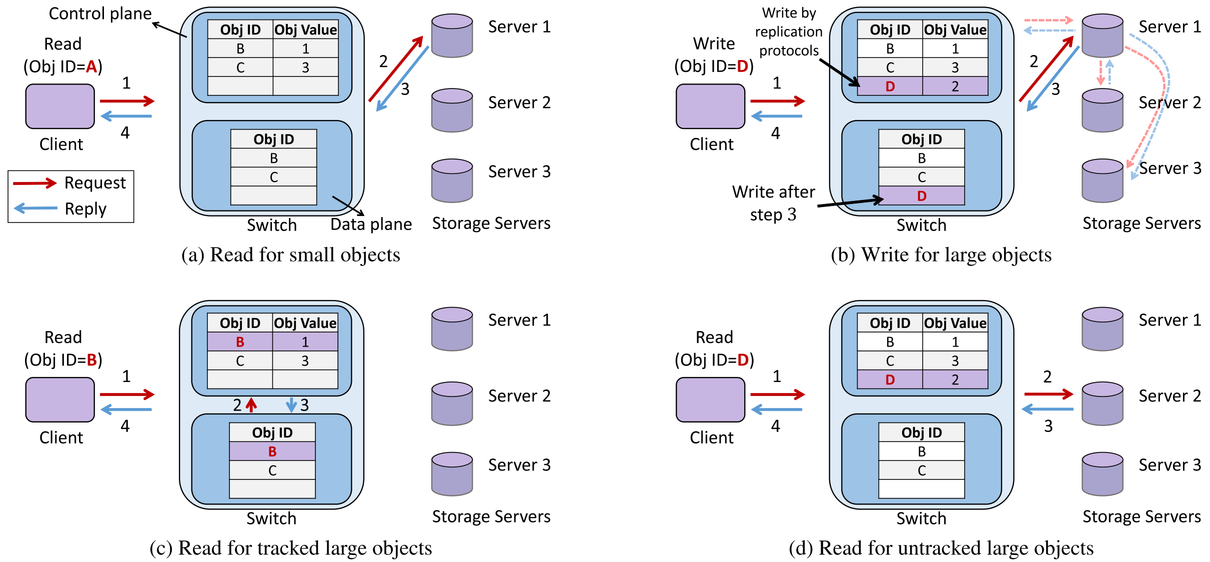


FIGURE 5. **Examples of request processing in NetStore.**

switch control plane. One difference is that the storage server compares the object size and the threshold to check whether the object is large or not.

2) WRITE REQUESTS

Upon receiving the write request, the switch data plane inserts the large object threshold into the NetStore header (line 7). Next, the switch removes the object ID from the large object list if the requested object exists in the list (lines 8-10). This is to prevent the client from reading the stale object data. Without this, the client may read the old object data in the switch control plane if there exists an ongoing write for the object. Furthermore, this also resolves cases where the object known as large becomes small with a new write because writes will not be forwarded to the switch control plane anymore. The switch data plane finishes write handling by forwarding the request to a storage server (line 16).

3) READ AND WRITE REPLIES

Upon receiving reply messages, the switch data plane inserts the object ID into the list if the object has the flag with *True* (i.e., large object) (lines 12-13). After that, the packet is forwarded to the client (line 16).

D. LARGE OBJECT TRACKING

To schedule read requests, the switch data plane needs to identify whether the object is large. However, we cannot know the object size in advance since the requested object size can be obtained in the storage server only. Therefore, we maintain the list of large objects in the switch data plane by leveraging register arrays. We only store the object ID in the list since the object data is returned by the switch control plane.

To track large object IDs, NetStore coordinates the switch data plane and storage servers. Specifically, we put the large object threshold into the `FLAG` field in the NetStore header when handling requests to deliver the threshold to storage servers. The threshold is stored in a data plane register and can be configured easily by the operator at run time. When sending replies, the storage server updates the `FLAG` field to *True* or *False* by checking the object size.

Algorithm 2 shows the object size identification process in the storage server. The only difference between read requests and write requests is where the object size comes from. Read requests obtain the object size from key-value stores (line 2) whereas write requests get the object size in the `SIZE` field in the header (line 9). Note that the size identification process for write requests applies only to the last packet of the request. We note that the switch data plane does not have enough computational capability to directly compare the object size in the `SIZE` field and the threshold metadata in the switch data plane. That is why we utilize storage servers for write requests as well even if the request contains the size information in the header.

As we have shown in Algorithm 1, the switch data plane inserts the object ID into the list when the `FLAG` field of the reply is *True*. It is not surprising for read requests because we can only know the exact object size after visiting the storage server. For writes, the write reply means the commitment of the ongoing write operation. Therefore, the insertion of object ID also means that the switch control plane and storage servers have the latest data. One limitation is that the first read request for a large object is forwarded to the storage server, but we believe that this is tolerable since the object is generally repeatedly requested.

E. EXAMPLES

We now show operational examples of request processing in NetStore. Figure 5 (a) shows how small requests are handled. Since object A is small, the large object list does not contain

the object ID. Therefore, the switch forwards the request to server 1, which is determined by replication protocols. Figure 5 (b) shows how the write request for object D is processed. To avoid reading the stale data of the next requests, the switch data plane first removes the object ID from the list if it exists. The switch control plane is updated during the write coordination. When the storage server commits the write by sending the reply, the switch data plane inserts the ID of object H into the list again. In Figure 5 (c), we can see that the read request for object B is processed in the switch control plane because the object is large and tracked by the switch data plane. Figure 5 (d) shows how NetStore handles read requests for untracked large objects. In this example, object D is maintained in the switch control plane, but the switch data plane does not have object D in the list. This can happen for various reasons like ongoing write operations, packet drops, and switch failures. In this case, the switch simply forwards the request to server 2 using replication protocols.

F. FAILURES HANDLING

1) DROPPED MESSAGES

Request messages can be dropped either in the switch or the NIC. Basically, the dropped messages can be detected and retransmitted using application-level timeout mechanisms. The primary concern is whether consistency is violated when a write message is lost. NetStore can preserve strong consistency because we only insert the object ID into the list when committing the write request. Let us assume that a write request for a large object is dropped at the packet buffer in the switch before reaching the storage server. In this case, since the switch data plane already removes the object ID from the list, read requests are forwarded to storage servers, thereby preserving strong consistency.

2) SWITCH FAILURES

Switch failures are rare in distributed storage. For example, the average number of switch failures per year is 1.1 [20]. When the switch fails, we can reboot the switch or replace it with a backup switch. During downtime, in-network request processing is unavailable because the switch cannot handle packets. However, it takes only tens of seconds to reboot the switch process. We note that the NetStore mechanism itself does not have an impact on switch failures because we only modify packet processing logic. We also note that the ToR switch is a single point of failure for any rack-scale system, regardless of our mechanism.

Similar to the case for dropped messages, the primary concern here is also consistency because the switch loses large objects in the control plane and the object tracking information in the data plane. Fortunately, clients do not read the stale data in the control plane, since the switch forwards large requests to storage servers where the latest data is stored. Furthermore, the switch can rapidly recover the information of large object IDs since the switch detects large objects when handling every read/write request. For the switch configuration, the configuration data is not lost because the data is stored in the disk, not memory.

3) SERVER FAILURES

Server failures are a generic problem in distributed systems, and NetStore does not cause a specific issue. Therefore, server failures can be handled by existing services like Apache Zookeeper [21] and etcd [22].

G. DEPLOYMENT ISSUES

1) SCALING TO MULTIPLE RACKS

While we basically target a storage rack where all storage replicas are co-located in the same rack, the current NetStore design also supports multi-rack deployment where each replica is located in different racks without modifications. This is because large objects in a rack are tracked by its ToR switch only. Each ToR switch can track large objects of each replica in a single replication group. One pitfall is that this makes switches track duplicate large objects. This may be resolved by making aggregation switches track object as well but is beyond the scope of this work. In addition, the critical path of a read request always contains the ToR switch connected to the storage server where the request object is stored. Therefore, aggregation switches only need to forward packets between the racks.

2) MULTIPLE REPLICATION GROUPS

There can exist multiple replication groups with different datasets in distributed storage. Similar to the multi-rack deployment, NetStore supports replicated storage with multiple replication groups. This is because, unless the requested object is large, NetStore does not specify the destination of requests regardless of replication groups. This means that NetStore is orthogonal to data partitioning techniques like consistent hashing [23] and request forwarding mechanisms based on the hash key.

H. DISCUSSION

We now discuss several issues related to the NetStore design.

1) CAPABILITY OF THE SWITCH CONTROL PLANE

Since the portion of large requests is small, the amount of requests that the switch control plane should handle is not large as well. However, the switch control plane might be saturated if a workload contains more large objects, making the switch the performance bottleneck. To avoid this, we can limit the portion of requests processed in the switch control plane by adjusting two system configuration parameters. One is the large object threshold and the other is the size of the large object list. Specifically, as we increase the threshold, more requests are forwarded to storage servers. In addition, if we reduce the large object list size, some requests do not visit the switch control plane even if the object is large because the object ID may not be stored in the list due to the lack of available slots.

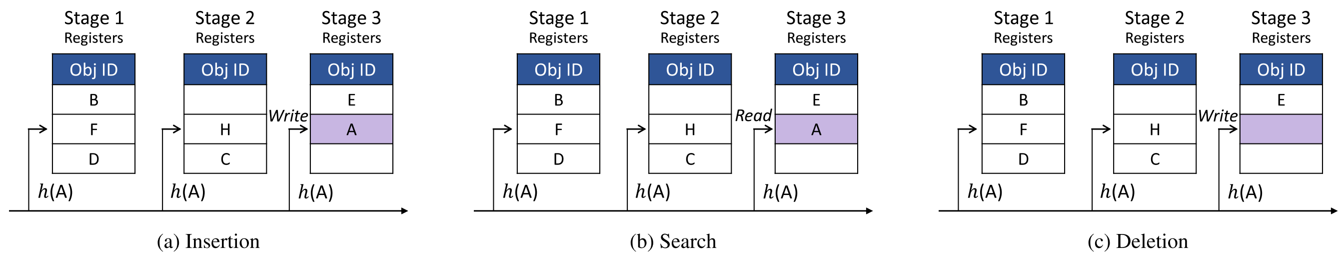


FIGURE 6. Insertion, search, and deletion in the multi-stage hash table to track large object IDs.

2) LARGE OBJECT THRESHOLD CONFIGURATION

We currently support a static configuration model where the system operator configures the threshold in the switch control plane by analyzing the collected workload traces. Typically, the threshold can be set to the object size at the tail (e.g., the 99th percentile request size). Since workloads can change over time, the threshold should be updated as well. To improve the maintainability, we may design a dynamic threshold configuration mechanism that adaptively changes the threshold value, and we currently investigate this for future work.

3) DYNAMIC LOAD BALANCING

NetStore relies on replication protocols to determine the destination server of read requests. For example, Hermes [2] simply picks a random replica. However, there exist various load balancing mechanisms like the join-the-shortest-queue (JSQ) and the power-of-k-choices, which can be implemented in the switch data plane [15]. Since NetStore does not have its own load balancing mechanism, we can integrate existing in-network load balancing mechanisms like RackSched [15]. Specifically, the switch basically forwards read requests to storage servers using the basic load balancing module but updates the destination to the switch control plane if the object is in the list.

V. IMPLEMENTATION

In this section, we describe how we implement a prototype of NetStore on the switch and the client-server application.

A. SWITCH CONTROL PLANE AND SERVERS/CLIENTS

1) THE SWITCH CONTROL PLANE

The switch control plane application is written in Python 3.9.12 using the Barefoot Runtime API. When we run the control plane application, the switch management module initializes the switch data plane with pre-configured table entries and register values. After that, the request processing module of the application waits for incoming requests.

2) CLIENT-SERVER APPLICATION

To measure the performance of distributed storage, we make an open-loop client-server application with the official Redis API. We use `pypacker` library [24]. The `pypacker` library allows us to define and manipulate packet headers easily and provides high-performance packet processing capabilities as well. The client can measure the throughput and latency by generating read and write requests at a given request sending rate. The server stores objects and serves read and write requests. The application performs packet segmentation/desegmentation in user space, allowing for custom processing of multi-packet requests in the switch.

B. DATA PLANE IMPLEMENTATION

Our switch data plane is written in $P4_{16}$ [16] and is compiled with Intel P4 Studio SDE 9.7.0 for Intel Tofino [14].

1) PACKET PROCESSING PIPELINE

The packet processing pipeline consists of the ingress pipeline, the egress pipeline, and the packet buffer between them. We implement our data plane modules in the ingress pipeline because NetStore determines the output port using a custom packet forwarding policy for in-network request processing. NetStore is a lightweight solution because it consumes only 6 M-A stages, which are less than the available resource budget of the switch. Our work uses only 6.82% of the switch memory, most of which is used to keep track of large object IDs.

2) MULTI-STAGE HASH TABLE WITH REGISTER ARRAYS

To track large object IDs in the switch data plane, we leverage register arrays. In an ideal implementation, we should provide a dedicated register slot for each object. However, the switch memory is limited and the size of register arrays must be determined at compile time. Therefore, similar to existing works [6], [7], we store object IDs in a hash table that uses the hashed object ID for indexing. The hash index is computed internally in the switch data plane. When using a hash table, we should address hash collisions because the hash table size may be smaller than the number of large objects that are requested at least once. Furthermore, since only a small amount of switch memory is allocated for each M-A stage, the maximum number of hash slots per stage is also limited. To address hash collisions and increase the number of hash slots, we implement the hash table using multiple M-A stages.

Figure 6 shows an example of our multi-stage hash table using 3 M-A stages. We use a single hash function, hence the object index is the same for every stage. To insert a new

object into the table, the switch finds a vacant slot across the tables. In Figure 6 (a), object A is inserted into the table in stage 3 since the hash slots in the previous stages are already occupied by objects F and H. Similarly, for search operations, the switch finds an occupied slot whose stored value is the same as the requested object ID (Figure 6 (b)). The deletion of object IDs is also similar, but it is different in that the switch removes the object ID from the slot, as in Figure 6 (c).

VI. EVALUATION

A. EXPERIMENT METHODOLOGY

1) TESTBED SETUP

To evaluate NetStore, we build a testbed consisting of 8 commodity servers, which are connected by an APS Networks BF6064X-T switch. The switch data plane is based on a 6.5 Tbps Intel Tofino switch ASIC [14]. The switch control plane is with an 8-core CPU (Intel Xeon D-1548@2.0 Ghz) and 32 GB of memory. The servers are with a 10-core CPU (Intel i5-12600K@3.7 Ghz), 32 GB of memory, and a single-port Ethernet NIC (NVIDIA MCX515A-CCAT ConnectX-5) where we set the link speed to 40 Gbps. The switch control plane runs Ubuntu 20.04 LTS with Linux kernel 5.4.0-generic. The servers also run Ubuntu 20.04 LTS, but are with Linux kernel 5.13.0-generic. Unless specified, 4 servers act as clients and the remaining 4 servers act as storage servers. The performance bottleneck is at storage servers.

TABLE 1. Configuration settings.

Configuration Item	Value
Switch ASIC	Intel Tofino
# of servers	8
Link bandwidth	40Gbps
Object ratio (small:large)	99.75:0.25
Large object threshold	200 KB

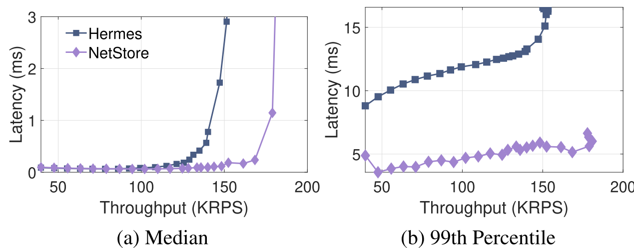


FIGURE 7. Throughput vs. latency.

2) WORKLOADS

To show the performance of reads and writes clearly, we use the read-only workload and the write-only workload. Our default workload is the read-only workload since NetStore mostly affects the read performance and many production workloads are ready-heavy with a 95:5 read:write ratio [6]. We also use 1M objects with 32-bit IDs and uniform distributions for the data access pattern. For objects, we consider a bimodal distribution with 99.75% of 128-B objects and 0.25% of 256-KB objects. This represents a typical workload that shows heavy-tailed object size distributions in the cumulative distribution function (CDF). Therefore, the portion of large requests is only 0.25%. The large object threshold is configured to 200 KB to consider 256-KB objects as large. The inter-arrival time between two consecutive requests of a client follows an exponential distribution. We summarize our settings in Table 1.

3) COMPARED SCHEME

We compare our work against Hermes [2], the state-of-the-art replication protocol that allows local reads and concurrent write coordination. We implement the basic read and write processing logic of Hermes. In our experiments, NetStore indicates 'Hermes with NetStore'. We can observe the impact of NetStore clearly by using Hermes as the baseline. To do this, we integrate NetStore with Hermes by making Hermes support the NetStore packet header and the object size identification mechanism.

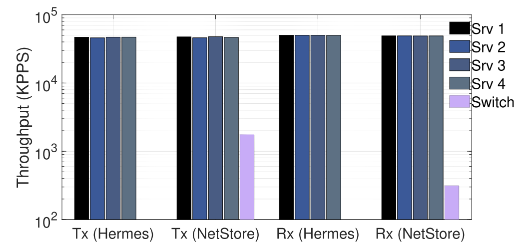


FIGURE 8. Breakdown of the load per server in NetStore.

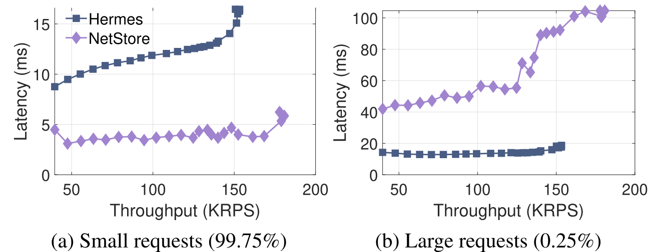


FIGURE 9. Performance across different object sizes.

B. EXPERIMENTAL RESULTS

1) THROUGHPUT VS. LATENCY

We first evaluate the latency as a function of the achieved requests per second (RPS). In this experiment, the clients generate requests to the storage servers, and measure the median latency and the 99th percentile latency at different throughput levels.

Figure 7 (a) shows the median latency for NetStore and Hermes. We can see that the latency of NetStore rapidly increases when the throughput reaches close to 152 KRPS. On the other hand, NetStore maintains a low median latency over 150 KRPS. The saturated throughput of NetStore is roughly 181 KRPS, which is better than Hermes by 1.19×. This is because NetStore provides extra computational power

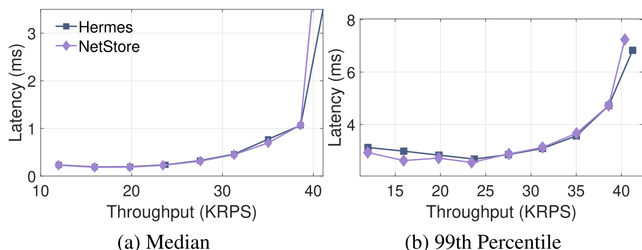


FIGURE 10. Throughput vs. latency (Write 100%).

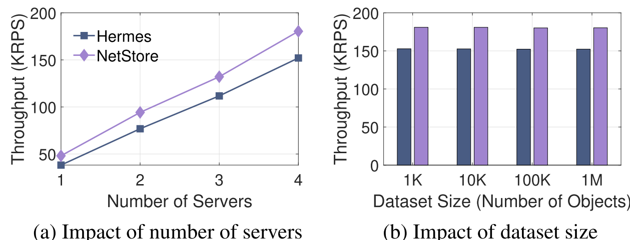


FIGURE 11. Scalability experiments.

for replicated storage through request processing in the switch control plane. In addition, storage servers can handle more requests per second, since most large requests are not processed by storage servers. Figure 7 (b) shows the result for the 99th percentile latency. We can also observe that NetStore achieves lower tail latency than NetStore. The performance gap is 2.91× when the achieved throughput is roughly 152 KRPS. This stems from that NetStore efficiently mitigates the head-of-line blocking caused by large requests. Our results imply that NetStore is an effective mechanism to improve the median and tail latency at higher throughput.

2) ROOT OF PERFORMANCE GAINS

Figure 8 and Figure 9 show the root of performance gains in detail. We break down the load in packets per second (PPS) across storage servers and the switch in Figure 8. Note that the Y-axis is presented on a logarithmic scale. The load for the storage servers seems similar in both Hermes and NetStore. However, NetStore has a slightly lower load for each storage server since large requests are forwarded to the switch control plane. In NetStore, the switch control plane processes large requests. Since read requests for the large object return multiple packets, the Tx throughput is higher than the Rx throughput.

Figure 9 (a) and (b) show the 99th percentile latency for small requests and large requests, respectively. Since large requests are generally processed in the switch, NetStore maintains stable tail latency for small requests, whereas Hermes shows a rapid growth curve. Meanwhile, NetStore degrades the tail latency of large requests as expected. As the throughput increases, the tail latency rapidly grows as well. This is because each large request faces a long latency in the switch control plane, which is crowded by multiple large requests. We believe that this trade-off between the performance of small requests and large requests is reasonable because we can significantly improve the performance of 99.75% of the requests by sacrificing only 0.25% of the requests. We also note that the performance degradation of large requests can be mitigated by limiting the portion of requests considered as large.

3) WRITE PERFORMANCE

Figure 10 shows the median latency and the 99th percentile latency across different throughput levels when the write ratio is 100%. In replicated storage, the write performance is always limited to the single server performance, since every storage server should update the requested object data. NetStore slightly degrades the write performance because replication protocols need to update the switch control plane when handling writes for large objects. However, since the portion of large requests is too small, the performance degradation is marginal. Furthermore, production workloads typically have 5% of write ratio, which means that the performance overhead caused by NetStore for write requests is small.

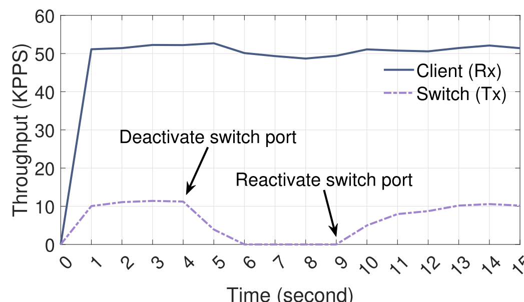


FIGURE 12. Performance under switch control plane failures.

4) SCALABILITY

Next, we evaluate the scalability of NetStore in the context of the number of storage servers and the dataset size. We vary the number of storage servers from 1 to 4 and also vary the dataset size from 1K to 1M. We then measure the saturated throughput. Figure 11 (a) shows the throughput according to the number of servers. We can see that NetStore and Hermes provide near-linear scalability. This is owing to the local reads enabled by Hermes. NetStore shows higher throughput than Hermes across all the number of servers, and this is because NetStore improves throughput through in-network large request processing. Figure 11 (b) shows that the throughput of NetStore is higher than Hermes, and the throughput is almost the same across the dataset sizes. This is because the dataset size itself does not affect the portion of requests.

5) PERFORMANCE WITH CONTROL PLANE FAILURES

We conduct an experiment to examine how NetStore behaves in switch control plane failures. In this experiment, we measure the Rx throughput of a client and the Tx throughput of the switch control plane. To simulate a control plane failure, we deactivate the switch port of the control plane at 4 seconds and reactivate the port at 9 seconds. In Figure 12, we can see a slight drop in the client throughput during the failure.

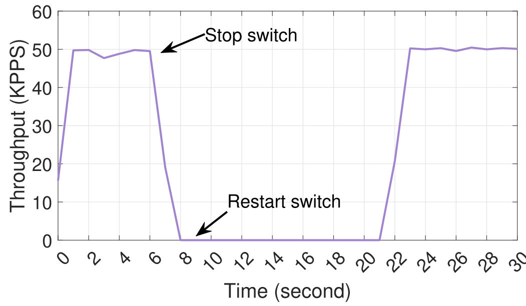


FIGURE 13. Performance under switch data plane failures.

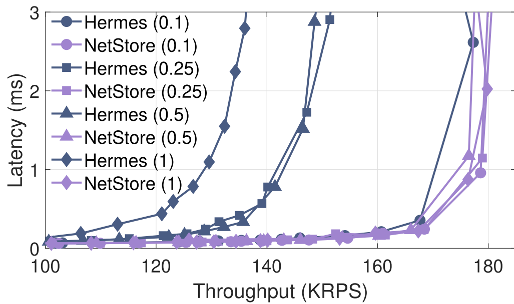


FIGURE 14. Impact of the portion of large requests (%).

This is because the switch control plane does not return the replies for large requests to the client. In a similar vein, we can also observe an increase in the client throughput when the switch port is reactivated. Meanwhile, the switch throughput is rapidly recovered when we reactivate the switch port.

6) PERFORMANCE WITH DATA PLANE FAILURES

We conduct an experiment to inspect the performance of NetStore under switch data plane failures. To simulate a switch failure, we stop the switch process at 6 seconds and restart it at 9 seconds. We measure network-level throughput by monitoring the network interface of a storage server. Figure 13 shows how NetStore deals with the data plane failure. When the switch is stopped, throughput drops to zero rapidly. However, throughput is recovered after roughly 14 seconds. The time to run the switch data plane depends on the switch architecture, and the NetStore mechanism has no impact on the time. We expect that the downtime can be decreased in the future with better switch architectures.

C. DEEP DIVE

1) IMPACT OF THE PORTION OF LARGE REQUESTS

We now inspect the impact of the portion of large requests. In this experiment, we vary the portion of large requests from 0.1% to 1%. Figure 14 shows the results, and we can know that NetStore is robust to the portion of large requests. It is easy to see that Hermes is sensitive to the portion of large requests. For example, as expected, Hermes shows the worst performance when 1% of the requests are large. On the other hand, NetStore shows similar performance regardless of the portion of large requests. This is because the switch has enough capability to handle the given portion of large requests.

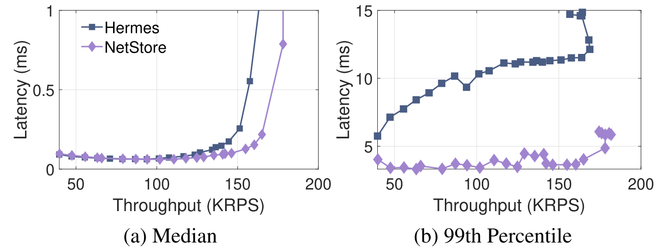


FIGURE 15. Performance with skewed workloads (Zipf-0.99).

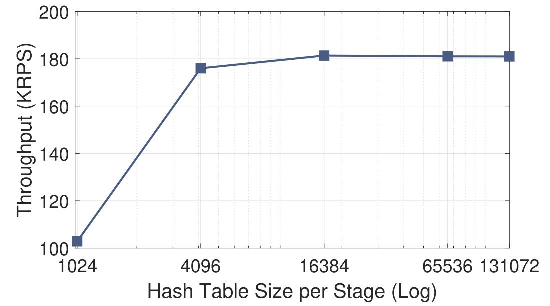


FIGURE 16. Throughput with different hash table sizes.

2) IMPACT OF ACCESS PATTERNS

We next evaluate the NetStore latency as a function of throughput for skewed workloads. To perform the experiment, the clients generate requests with a zipfian distribution with the parameter 0.99. Similar to an existing work [9], we use the zipfian distribution for small objects only to avoid pathological cases where the most frequently accessed object is a large object, which is rare in practice. Figure 15 (a) shows the median latency. We can see that NetStore provides better throughput than Hermes even with skewed workloads. In Figure 15 (b), we can see that NetStore results in better tail latency than Hermes across all the measured throughput levels.

3) IMPACT OF HASH TABLE SIZE

We now measure the saturated throughput by varying the hash table size per each M-A stage. This is to inspect how much of the switch memory is required to track large objects in the switch data plane. Figure 16 shows the throughput as the hash table size grows. We can see that NetStore only requires about 4096 hash slots to achieve performance close to the best performance. 4096 hash slots with 3 M-A stages consume 48 KB of memory. By default, we use 128K hash slots per stage to accommodate higher request rates. Note that we can increase the hash table size using more stages, and the maximum number of stages depends on switch ASICs, not the NetStore mechanism.

VII. RELATED WORK

We now discuss existing works related to NetStore.

A. REPLICATION PROTOCOLS

Replication protocols can be categorized into leader-based protocols and leaderless protocols. The primary backup (PB)

[4] is the basic leader-based protocol in that the stable leader handles all requests, while followers exist for only backup. CR [3] improves the performance by using a dedicated server for read processing and replica chaining. CRAQ [8] allows local reads by maintaining the object cleanness in each storage server. Hermes [2] is the state-of-the-art leaderless replication protocol that allows local reads and concurrent write coordination. Thanks to these features, Hermes provides the best performance between the replication protocols. NetStore is orthogonal to the replication protocol, since NetStore only addresses in-network read processing and do not modify the replication process.

B. IN-NETWORK SOLUTIONS FOR STORAGE SYSTEMS

Several works have investigated how to improve the performance of distributed storage using programmable switches [15], [25], [26], [27]. NetCache [25] shows that caching objects in the programmable switch can resolve the load imbalance problem. However, it can cache only tiny objects and occupy over half of the switch memory. IncBricks [26] and PMNet [27] also cache objects in hardware. These work use FPGA-based hardware while NetCache utilizes commodity hardware. All the works do not mitigate the performance overhead of large requests because they assume that every object has the same and small size. RackSched [15] shows that we can balance request loads through in-network load balancing. Although RackSched can mitigate the head-of-line blocking, it still makes large requests processed in the storage server. NetStore is complementary to RackSched because RackSched concerns load balancing whereas NetStore relies on replication protocols for load balancing.

Harmonia [7] provides near-linear scalability by detecting read-write conflicts in the network for replicated storage. NetLR [6] moves the entire replication function into the network by leveraging the switch as an in-network replication coordinator. Since the above two works concern the replication function, they are orthogonal to NetStore, which only concerns in-network read processing. In addition, one distinct contribution of NetStore is to discover the acceleration opportunity of the switch control plane, not relying on the switch data plane.

C. RPC ACCELERATION

Storage access between clients and servers is typically done by Remote Procedure Calls (RPCs), and there are recent works trying to accelerate RPC calls. ALTOCUMULUS [28] uses direct messaging between NIC and registers to mitigate the RPC scheduling overhead. nanoPU [29] paves a fast data path between the NIC and applications to bypass the cache and memory hierarchy. Similarly, RPCValet [30] uses shared caches to bypass slow PCIe paths. Meanwhile, eRPC [31] accelerates RPC applications by optimizing several common performance degradation factors by software techniques. NetRPC [32] is a generic RPC system for in-network solutions that provides a lightweight software interface while maintaining comparable performance to existing in-network solutions. mRPC [33] is an RPC system that provides RPC marshalling and policy enforcement as a service, not a library linked to applications to achieve manageability and efficiency at the same time. Since NetStore is an in-network solution and the above works are server-side solutions, they are complementary.

VIII. CONCLUSION

This paper presented NetStore, an in-network storage accelerator for replicated storage to provide high throughput and low latency without the performance overhead of large requests. NetStore leverages the programmable switch as an in-network request processor to mitigate the head-of-line blocking and reduce server resource usage of large requests. We showed the co-design of the switch control plane and the switch data plane to overcome the strict resource constraints of commodity switch ASICs. We have implemented a NetStore prototype on an Intel Tofino switch and conducted testbed experiments to demonstrate that NetStore can accelerate replicated storage. We believe that this work can contribute to the research community by demonstrating that leveraging the architectural opportunities of switch hardware in a holistic manner leads to significant performance acceleration for heavy-tailed storage workloads.

For future work, we have two directions as follows. First, the current NetStore design requires network operators to update the large object threshold manually. If we automatize the threshold update by designing an adaptive threshold update mechanism, the maintainability of the system can increase dramatically. Second, NetStore currently uses a random load balancing algorithm to select a destination storage server. However, there exist better load balancing algorithms like the join-the-shortest-queue algorithm, which can be implemented in the switch data plane. Therefore, integrating a new in-network load balancing mechanism and the NetStore pipeline may be an attractive direction to increase the performance gain of NetStore.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments and constructive feedback and also would like to thank Jiyoon Bang for her help in drafting the evaluation section.

REFERENCES

[1] *Redis Key-Value Store*. Accessed: Jul. 26, 2023. [Online]. Available: https://redis.io/
[2] A. Katsarakis, V. Gavrielatos, M. R. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, "Hermes: A fast, fault-tolerant and linearizable replication protocol," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 201–217.
[3] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. USENIX OSDI*, San Francisco, CA, USA, Dec. 2004, pp. 91–104.
[4] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proc. ICSE*, Washington, DC, USA: IEEE Computer Society Press, Oct. 1976, pp. 562–570.

[5] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[6] G. Kim and W. Lee, "In-network leaderless replication for distributed data stores," *Proc. VLDB Endowment*, vol. 15, no. 7, pp. 1337–1349, Mar. 2022.

[7] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin, "Harmonia: Near-linear scalability for replicated storage with in-network conflict detection," *Proc. VLDB Endowment*, vol. 13, no. 3, pp. 376–389, Nov. 2019.

[8] J. Terrace and M. J. Freedman, "Object storage on CRAQ: High-throughput chain replication for read-mostly workloads," in *Proc. USENIX ATC*, 2009, p. 11.

[9] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores," in *Proc. USENIX NSDI*, Boston, MA, USA, 2019, pp. 79–94.

[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Modeling Comput. Syst.*, Jun. 2012, pp. 53–64.

[11] M. Blott, L. Liu, K. Karras, and K. Vissers, "Scaling out to a single-node 80 Gbps memcached server with 40terabytes of memory," in *Proc. USENIX HotStorage*, 2015, pp. 8–12.

[12] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at Facebook," in *Proc. USENIX NSDI*, Berkeley, CA, USA, 2013, pp. 385–398.

[13] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *Proc. USENIX NSDI*, Boston, MA, USA, MA, Feb. 2019, pp. 345–360.

[14] *Tofino Programmable Switch*. Accessed: Jul. 26, 2023. [Online]. Available: https://github.com/barefootnetworks/Open-Tofino

[15] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin, "RackSched: A microsecond-scale scheduler for rack-scale computers," in *Proc. USENIX OSDI*, Nov. 2020, pp. 1225–1240.

[16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "p4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[17] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook," in *Proc. USENIX FAST*, Santa Clara, CA, USA, Feb. 2020, pp. 209–223.

[18] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing SoC accelerators for memcached," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, Jun. 2013, pp. 36–47.

[19] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 99–110.

[20] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM Conf.*, Aug. 2011, pp. 350–361.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX ATC*, 2010, p. 11.

[22] *ETCD Key-Value Store*. Accessed: Jul. 26, 2023. [Online]. Available: https://etcd.io/

[23] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory Comput. (STOC)*, 1997, pp. 654–663.

[24] *Pypacker: The Fastest and Simplest Packet Manipulation Lib for Python*. Accessed: Jul. 26, 2023. [Online]. Available: https://gitlab.com/mike01/pypacker

[25] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 121–136.

[26] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," in *Proc. ACM ASPLOS*, New York, NY, USA, 2017, pp. 795–809.

[27] K. Seemakhupt, S. Liu, Y. Senevirathne, M. Shahbaz, and S. Khan, "PMNet: In-network data persistence," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 804–817.

[28] J. Zhao, I. Uwizeyimana, K. Ganesan, M. C. Jeffrey, and N. E. Jerger, "ALTOCUMULUS: Scalable scheduling for nanosecond-scale remote procedure calls," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2022, pp. 423–440.

[29] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown, "The nanoPU: A nanosecond network stack for datacenters," in *Proc. USENIX OSDI*, Jul. 2021, pp. 239–256.

[30] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-driven tail-aware balancing of μs-scale RPCs," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 35–48.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter RPCs can be general and fast," in *Proc. USENIX NSDI*, 2019, pp. 1–16.

[32] B. Zhao, W. Wu, and W. Xu, "NetRPC: Enabling in-network computation in remote procedure calls," in *Proc. USENIX NSDI*. Boston, MA, USA: USENIX Association, Apr. 2023, pp. 199–217.

[33] J. Chen, Y. Wu, S. Lin, Y. Xu, X. Kong, T. Anderson, M. Lentz, X. Yang, and D. Zhuo, "Remote procedure call as a managed system service," in *Proc. USENIX NSDI*. Boston, MA, USA: USENIX Association, Apr. 2023, pp. 141–159.

GYUYEONG KIM (Member, IEEE) received the B.S. and Ph.D. degrees in computer science from Korea University, South Korea, in 2012 and 2020, respectively. In 2022, he joined as a Faculty Member with Sungshin Women's University, Seoul, South Korea, where he is currently an Assistant Professor with the Department of Computer Engineering. Before joining the Sungshin Women's University, he was a Research Professor with the Future Network Center, Korea University. His research interests include networked systems, in-network computing, scriptable network stack, and HW-SW co-design.