JoiNS: Meeting Latency SLO with Integrated Control for Networked Storage

Hao Wen[1], Zhichao Cao[1], Yang Zhang[1], Xiang Cao[2], Ziqi Fan[1], Doug Voigt[3], and David H.C. Du[1]

[1]University of Minnesota, Twin Cities
[2]Grand Valley State University
[3]HP Enterprise

***Abstract*—Meeting latency SLOs (Service Level Objectives) in a networked storage environment is essential while challenging. In this environment, a storage request has to go through client I/O stacks, dynamically changing networks, and the storage system attached to a server. Its response also has to traverse all the way back to the client. Along this long I/O path, any of these components can become congested. The behavior of one component may affect the performance of the others. Isolated control on each component is not effective to meet latency SLOs of storage requests. In this paper, we propose and implement JoiNS, a system trying to guarantee latency SLO for applications that access data on a remote networked storage. JoiNS carefully considers all the components along the I/O path and controls them in a coordinated fashion. JoiNS has both global network and storage visibilities with a logically centralized controller which keeps monitoring the status of each involved component. JoiNS coordinates these components and adjusts the priority of I/Os in each component based on the latency SLO, network and storage status, time estimation, and characteristics of each I/O request. We integrate Software Defined Network(SDN) into our system to coordinate with storage. Our evaluation shows JoiNS can achieve up to 6X speedup in this networked storage environment with various loads of background traffic.**

I. Introduction

In recent years, networked storage has become a popular type of storage. In this environment, storage is connected to clients through network. Cloud storage [1], [4], [2], [3], datacenter SAN and NAS, object storage are typical networked storage. When accessing networked storage, an I/O request will go through the client side I/O stacks, transit through the network, traverse the storage servers and finally be served by storage devices like disks. The response of the request also has to go through these components on the reverse path back to the client. This long I/O path and the diverse components along the path result in increased end-to-end management complexity.

Such increased complexity makes meeting latency SLOs (Service Level Objectives) harder in this networked storage environment. First, it requires all components along the I/O path to react on incoming I/Os in a coordinated fashion. Second, the status of each component is dynamically changing. Third, each component treats I/O requests differently since they have different semantics.

Currently, there are several mechanisms trying to ensure SLOs for network and storage individually. However, such individual control may not be effective and even bring in more issues. For example, duplicating I/O requests and returning the fastest response from multiple storage replicas may reduce the latency in storage access time but cause congestion on network by creating more network traffic. On the other hand, a solution may be obtained by carefully considering all components along the I/O path with a systematic control that efficiently utilizes available resources at some components while tolerating some performance degradation at other components. For example, we can allow some I/Os to be served with priority in the congested network so that they can be processed by storage devices sooner and returned to the client in time. Studies like [25], [26], [29], [23] consist of control on both network and storage when trying to ensure SLO. However, further study is still needed on how to coordinate network and storage and fully utilize their characteristics. We believe that a systematic way of coordinating all of the involved components is necessary in order to ensure SLO in this environment.

In this paper, we introduce JoiNS, a system that coordinates different components along the I/O path to meet latency SLO in a networked storage environment. We bridge the gaps between different components by deploying a logically centralized controller to orchestrate the control to each component. The controller has a global view of the network and storage status. It keeps collecting information of the current network, and storage status and estimates the latency at network and storage respectively for each I/O request. It determines whether to control I/Os based on the latency SLO, network and storage status, time estimation and I/O characteristics. The controller interacts with enforcers at client, network and storage nodes to coordinate their actions. Enforcers along the I/O path will adjust the priorities of I/Os accordingly. Software Defined Network(SDN) [10], [8], [14], [17] is integrated into JoiNS as part of the network enforcers to coordinate with storage. To avoid possible scalability issues, this logically centralized controller may have delegates on each node but function as a whole. When controlling I/O requests and responses, we distinguish reads from writes. We utilize the asymmetry property in read and write I/O packet size to better ensure the SLO of an I/O request with less penalty incurred on other traffic.

In this paper, the following contributions are made:

- We identify the need to consider all the components along the I/O path from client to storage to ensure latency SLO.
- We design controller-based mechanisms to monitor the

status of network and storage globally, estimate the latency of I/O requests and guide the control along the I/O path.
- We design an approach to control I/O packets with little overhead based on the asymmetry property in read and write.
- We build a real system to coordinate clients, network, and storage, and demonstrate the effectiveness of JoiNS in ensuring latency SLO.

II. Related Work

Storage QoS or SLO has been studied for a while. Recently, as I/O stacks become more complicated, some research start to investigate QoS guarantee under different I/O stacks, including virtual machines [27], containers [16], and networked storage [29]. In this paper, we focus our discussion on the existing studies that try to involve both network and storage in networked storage environment. Recently, several researchers see the essence of including network when considering storage QoS, or vice versa. IOFlow [25] is an architecture that can enforce high-level flow policies. It allocates bandwidth for a flow from a particular VM to certain storage shares or routes an I/O traffic through a sanitization layer. In practice, it exercises control on SMB client and SMB server (SMB is a network file sharing protocol [9]), but it does not have control on network that is between SMB client and SMB server. A congested network may break the SLO compliance they are trying to meet. JoiNS shares the same goal of meeting application's SLO requirements, but we tackle the scenario that both network and storage may become congested. In addition, JoiNS aims at solving the problem when resources are in shortage, especially at the boundary of getting close to congestion situation. IOFlow does not aim at this problem specifically. They share resources evenly among VMs whose requirements cannot be met.

sRoute [23] extends IOFlow's routing functions and is able to forward I/Os from over loaded servers onto less loaded servers. The forwarding is determined based on the queue size at each of the storage servers. The rerouting may change the traffic on network but it does not consider the network status.

PriorityMeister [29] tries to meet end-to-end tail latency SLOs by automatically and proactively configuring workload priorities and rate limits. It assumes the system has full visibility and control over all workloads which may not be the truth in a networked storage environment. There is always traffic beyond our control including workloads from other users or data centers. In JoiNS, we monitor those traffic at real time. We react to the network and storage status change as workloads fluctuate. In JoiNS, priorities are dynamically assigned to each I/O rather than statically assigned in the granularity of workload.

III. Motivation and Challenges

There are many mechanisms on alleviating the performance degradation of a single component. Controlling a single component and ignoring other components may not help the overall performance. For example, people may take the redundancy-based approach to tackle the long tail latency issue [28]. In case of storage congestion, the redundancy-based approach may send redundant requests to multiple replicas of the same object to seek for light loaded storage. This will bring about additional cost on network and may saturate some network routes. In fact, it is essential to be aware of the status of each component before taking any control actions.

Meanwhile, more performance improvement may be achieved by controlling multiple components together. Considering that in a networked storage environment, the network is congested and the latency SLO of a client is violated. Classical techniques like RED [12] and ECN [11] are used to alleviate network congestion. But they ignore the storage status. If the storage is light-loaded, we may let I/Os be served with priorities in the network and go to the light-loaded storage earlier by prioritizing those I/Os. Therefore, in a system involving multiple components like client, network and storage, a solution trying to ensure latency SLO should coordinate all components along the I/O path.

However, there are some challenges in coordinating components along the I/O path to ensure SLO. First, in order to know when and where we should exercise control, we need to have a global view of the status of each component along the I/O path. However, it is challenging to acquire this global view since network and storage nodes are often remotely located and geographically distributed. Second, coordination among multiple components requires understanding of the interactions among them and the impact of controlling one component over other components.Third, in order to ensure latency SLO, it is required that a control policy knows the exact latency requirement. How to inform each I/O request and these components of the SLO is a problem. Fourth, any control imposed on the I/O requests will induce additional overhead on the performance. We need a control mechanism to ensure SLO with less overhead.

IV. Architecture

JoiNS ensures the latency SLO of I/O requests by coordinating different components along the I/O path.

A. System Design

Figure 1 shows the architecture of JoiNS. Client, network, and storage are monitored by a logically centralized controller. The controller keeps collecting the status data of each network and storage node via the **Status Monitor** module. These data are used to estimate the time needed for each I/O request at network request path, network return path, and storage in **Time Estimation** module. By comparing the estimated time with the required SLO, it can determine whether to control I/Os along the path in **Policy Enforcement** module.

Each component along the I/O path has an enforcer to control I/Os. The client enforcer controls admission of I/Os into network. Once an I/O request is delivered by NIC as I/O packets, they will enter their request path of network. The network enforcer at each network node can identify the packets that need control. Each enforcer implements differentiated scheduling through queues. Queuing rules are configured by

the controller to dispatch I/O packets with different priorities into different queues. The priority of an I/O packet is set by each network enforcer according to the control decision made by the controller. In storage, once I/O packets arrive, they will first be reorganized as I/O requests. These I/O requests then will be processed by the storage enforcer. A differentiated scheduler is also implemented to differentiate I/O requests. Finally, they get processed by the storage device. The queuing rules in the storage enforcer are configured by the controller. Since it is I/O response generated by the storage that is delivered on the network return path, the control information related to an I/O request should be embedded into the response by the storage enforcer. On the return path, I/O packets will be matched again by network enforcers for possible control.

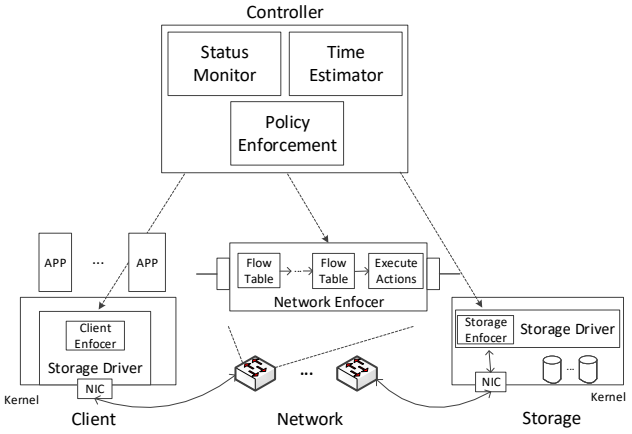


Fig. 1: System Architecture of JoiNS.

B. Status Detection

We design an algorithm for the status monitor module to have a global view of the network and storage status. We call it "probe and test". Considering any control imposed on I/Os will generate overheads, we will try to reduce the control. A control process will be wasteful when there is no congestion or little congestion in network and storage. They are also useless when the system is totally congested. Therefore, we need to determine the congestion level of the system first during runtime. The status monitor module periodically sends active probes, which are specially designed I/O requests, to the storage and receives the responses. These probes collect data that can reflect network and storage status at real time.

The status monitor module will send out 3 probes in each round of probing: one read, one write and one storage query. To minimize the load imposed on network and storage, each read or write probe only contains the I/O request with the typically smallest I/O size, e.g., 4KB and they will not be actually processed by storage. Once received, the corresponding responses will be returned immediately by storage. The size of a read/write probe and its response will follow the actual size of the read/write request/response strictly. Each probe will be timestamped on entering the network, entering the storage, exiting storage, and arriving back at client. They are denoted as T_1, T_2, T_3, T_4 respectively. Hence the time spent on the request path and return path of the network for a read probe are calculated as $T^r_{rq} = T_2 - T_1$, $T^r_{rt} = T_4 - T_3$. We also have T^w_{rq}, T^w_{rt} for write probes. The storage will send back the current queuing time T_q in storage after receiving the storage query probe. All probes are assigned default priorities along the way.

By using these data, the controller can have an estimation of latency t_{est} for each I/O request. The controller will determine the congestion level for each I/O request by comparing its SLO (D) with the estimated latency. We set a congestion factor β $(0 < \beta < 1)$ to define a safe zone for the latency. If $t_{est} \leqslant \beta D$, it means the current latency is safe and the system is **not congested** for this request. Hence, there is no need to do anything. If $\beta D < t_{est} \leqslant D$, it indicates that the system is **close to congestion** and there is a danger that the SLO will be violated. In this case, the controller determines to control this I/O request along the path. If $t_{est} > D$, it means the system is **fully congested**. In this case, we throttle the client. This congestion factor β can be adjusted by users of JoiNS.

C. Time Estimation

The network latency of an I/O can be characterized as the sum of transmission delay, propagation delay and queuing delay. We assume there are k hops from one end to the other, and the transmission speed on each link is $g_i (i \leq k)$. The MTU size of each link is $MTU_i (i \leq k)$. According to the packet switching theory [24], the latency for transferring x KB data from one end to the other is:

$$L = \frac{x}{g_{min}} + P + Q + \sum^{k}[\frac{MTU_i}{g_i} - \frac{MTU_i}{g_{min}}]$$

Where g_{min} is the minimum of all g_i. P is the propagation delay. Q is the queuing delay. Apparently, L is linear to x, and can be rewritten as $L(x) = a \cdot x + b$. a is the reciprocal of g_{min}. It can be directly calculated from g_{min} or calculated from multiple $(x, L(x))$ data points. b reflects the current network status (queuing delay plus other constants).

For a read probe, we have $T^r_{rq} = a_{rq} \cdot S_{\text{req}} + b_{rq}$, $T^r_{rt} = a_{rt} \cdot S_0 + b_{rt}$, where S_{req} is the packet size of a read request and S_0 is the I/O size of the probe. For an m KB read request, we can estimate the time on the network request path t^r_{rq} and the time on the network return path t^r_{rt}:

$$t^r_{rq} = a_{rq} \cdot S_{\text{req}} + b_{rq} \tag{4.1}$$

$$t^r_{rt} = a_{rt} \cdot m + b_{rt} \tag{4.2}$$

With each collected T^r_{rq} and T^r_{rt}, we can calculate the current network status b_{rq} and b_{rt}, and use them to calculate equation (4.1) and (4.2). Similarly, we can calculate t^w_{rq} and t^w_{rt} for write requests from T^w_{rq} and T^w_{rt}. Based on the service center model, the estimated time in storage can be calculated as $t^r_s = T_q + \frac{m}{Bandwidth_r}$ and $t^w_s = T_q + \frac{m}{Bandwidth_w}$. $Bandwidth_r$ and $Bandwidth_w$ are the storage bandwidth for read and write respectively.

So we can calculate the estimated latency for an m KB read request t^r_{est} and for an m KB write request t^w_{est}:

$$t^r_{est} = t^r_{rq} + t^r_{rt} + t^r_s \tag{4.3}$$

$$t^w_{est} = t^w_{rq} + t^w_{rt} + t^w_s \tag{4.4}$$

We will discuss how we control I/Os with these estimated latency in Section IV-E.

D. Distinguish Read from Write

Our special control of I/Os is based on the asymmetry property existing in read and write I/O packet size. In a networked storage environment, the I/O requests and their responses are encapsulated in I/O packets on network. Traditionally, network functions like routing and switching do not treat these I/O packets differently and hence ignore the intrinsic characteristics inside storage I/Os.

On the network request path, a read request contains only I/O request while a write request contains the data to be written to the storage. On the other hand, on network response path, a read response contains the data that the client requests while a write response is just a notification of success or failure of the write. Taking an example of iSCSI, a read request for reading 4KB of data has only 48B data including a 16B SCSI command encapsulated in a single network packet. On the other hand, a write request writing 4KB data includes not only a write SCSI command but also the 4KB data to be written. Since the size of data to be written is typically much larger than one MTU size, a write request will be broken into multiple network packets. On the response path, this asymmetry reverses. The read response contains 4KB data spanning across multiple network packets. The response of that write request is a 48B iSCSI response indicating write success or failure.

Since we want to ensure the SLO of each I/O request, we may choose to prioritize all network packets composing that I/O request and its response. However, prioritizing these I/O packets is sure to delay other packets. Considering the asymmetry in I/O read and write, we distinguish reads from writes. On the request path, we choose to prioritize read requests. On the return path, we prioritize write responses due to their smaller size.

E. Coordination

JoiNS coordinates client, network and storage to ensure the latency SLO of I/Os.

In order to coordinate network and storage, how network can understand the interactions with storage and vice versa becomes the first problem. In a networked storage environment, an I/O request is delivered in one or multiple network packets on network. They are then extracted out from network packets at storage and the network packet headers are discarded. Considering network preserves all storage information while storage loses network information, all control information sent to storage should be incorporated in the command (e.g., SCSI command) of the I/O request and all control information sent to network can be incorporated in network headers.

Once the controller determines the congestion level for an I/O through probe and test algorithm, it will determine whether to control this I/O. If the system is not congested, this I/O does not need any control. The client enforcer will admit that request directly into network. If the system is tested to be close to congestion, the controller will control this I/O. The client enforcer will mark that I/O in the corresponding network packet headers and the storage command. If the system is tested to be fully congested, the client enforcer will also deliver that request into network and let the client be throttled by TCP congestion control.

Once I/O requests are admitted into network, the priorities of I/Os will be adjusted at each network node and the storage node. If an I/O is marked by the controller, network enforcers can recognize it by matching the corresponding I/O packets against the queuing rules. According to the matched rules, a priority will be set for each I/O packet. Then, each packet will be dispatched to the queue matching the priority. The storage enforcer will also set the priority of the I/O request extracted from the received I/O packets, and dispatch it to the queue matching the priority.

V. Implementation

Our implementation is based on iSCSI. Our storage is connected through Ethernet and IP networks to clients. A client accesses its storage via an iSCSI initiator to the iSCSI target (networked storage device) at a remote location.

Client enforcer is responsible for admitting I/O requests into network. We implement and insert a hook inside iSCSI initiator. If the controller determines that a read request needs further help, the client enforcer will tag the packet of that read request with DSCP flags which can be recogonized by network enforcers. It also sets a miscellaneous byte of iSCSI CDB [20] such that this request can be recognized by storage enforcer. If a write request needs further assistance, it only sets the miscellaneous byte of iSCSI CDB. After the iSCSI target receives it, the storage enforcer will tag the response of that write request with DSCP flags so that it can be recognized on the return path of the network.

Network enforcer is responsible for initiating and dynamically configuring priority queues in programmable switch. Pioneers like [18], [15], [13] have tried to optimize I/Os on network devices with network functions, e.g., data deduplication [22], [7], etc. Our network enforcers also control I/Os in network. In our prototype implementation, we utilize Openflow virtual switch [17], [5] (OVS) as the programmable switch (an SDN implementation). Please note that JoiNS does not rely on Openflow-capable switch. Legacy switches which provide programmability interface can also be integrated into JoiNS. The OVS version we use in the prototype is 2.0.2, and the OpenFlow protocol follows version 1.3.

In our implementation, we attach two queues on each output port of an OVS. Both queues attached to a port have the same maximum rate equal to the link rate of that output port, but have different minimum rates. A queue with a higher minimum rate has a higher priority. We configure one queue with 0 minimum rate as regular queue, and the other queue with minimum rate equal to link rate as fast queue. When the network is light-loaded, we only use the queue with minimum rate of 0. Because there is no contention from the other queue, it functions as if there is no priority differentiation at all and can take the full link bandwidth of the output port.

We implement a control application that will determine the priorities of I/O packets and install flow entries into Open vSwitches. Each OVS will first match I/O packets against the flow entries to check whether the DCSP flag is marked. If an I/O packet is marked, it means the packet needs control. Then the packet will be forwarded to other flow tables inside the switch to match flow entries further to determine its priority inside this network node. Finally it will be forwarded to the fast or regular queue attached to the output port.

Storage enforcer is responsible for differentiating and scheduling I/O requests. Our current implementation inserts a hook inside iSCSI target. Once an I/O request is admitted, the iSCSI target checks the miscellaneous byte in iSCSI CDB. If it is set, it determines the priority of the request and dispatches it to the fast or regular queue. In addition, the iSCSI target will take responsibilities of marking write responses by tagging the packets with DSCP flags if the iSCSI CDB miscellaneous byte is set in the corresponding write requests.

VI. Evaluation

This section evaluates JoiNS. Part A presents the setup of experiments. We demonstrate the ability of JoiNS in meeting latency SLO in a networked storage environment in Part B. Part C evaluates the cost of prioritizing only read requests and write responses utilized in JoiNS.

A. Experiment setup

Our testbed comprises 5 servers, each with 24 Intel Xeon 2.40GHz E5-2620 v3 cores and 64GB of memory. Each server has four 1Gb/s Broadcom NetXtreme BCM5720 NIC ports, connected to an HP ProCurve 5406zl switch. They are all running Ubuntu 14.04. One server acts as a client that sends I/O requests to storage. Two servers act as storage nodes. One of them serves as iSCSI target and provides storage. The other one serves as the storage proxy. They use one HDD volume with maximum bandwidth of 120MB/s as the backend. The remaining two nodes serve as SDN switches with the link speed of 1Gb/s.

Datasets. We evaluate our system using a collection of real block I/O [19] and synthetic storage traces. Table I summarizes all the workloads we use in the evaluation. For real block I/O traces (Workload A - E), we also show the squared relative standard deviation of the interarrival time, which describes the burstiness of the workload. A higher squared relative standard deviation indicates more bursty arrival patterns. With different interarrival variances, we show our system can adapt to different burstiness.

Policies. In our experiments, we use 5 approaches to understand different aspects of the performance of JoiNS. *JoiNS* is our primary mechanism that tries to ensure latency SLO with little overhead. It is able to adapt to network and storage status change. In our experiment, we set the congestion factor β to be 0.8 when the controller determines the congestion level for each I/O. In the *Legacy* system, all components are in default configurations. That is, FIFO in network and storage. In *Pri_all* approach, we do not judge the congestion level for I/Os but prioritize all read requests and write responses of an application. In *Pri_both* approach, if an I/O needs to be prioritized, both the request and the response will be prioritized. We use this approach to evaluate the performance of distinguishing read from write. We implement the policy in the PriorityMeister [29] paper as *PM* to evaluate the latency SLOs for multiple latency sensitive workloads. *PM* works in the system which has full knowledge and control of workloads. It analyzes the representative trace proactively for each workload and applies the r-b pairs (rate and token bucket size) to the workloads which enable workloads run in an unfettered manner. It uses a greedy algorithm to pick a priority ordering of the workloads. In the experiment, we first analyze the trace we are replaying and generate the r-b pairs used in token bucket for each workload. With predefined latency SLO, we use the prioritizer algorithm described in the paper to configure the priorities of workloads.

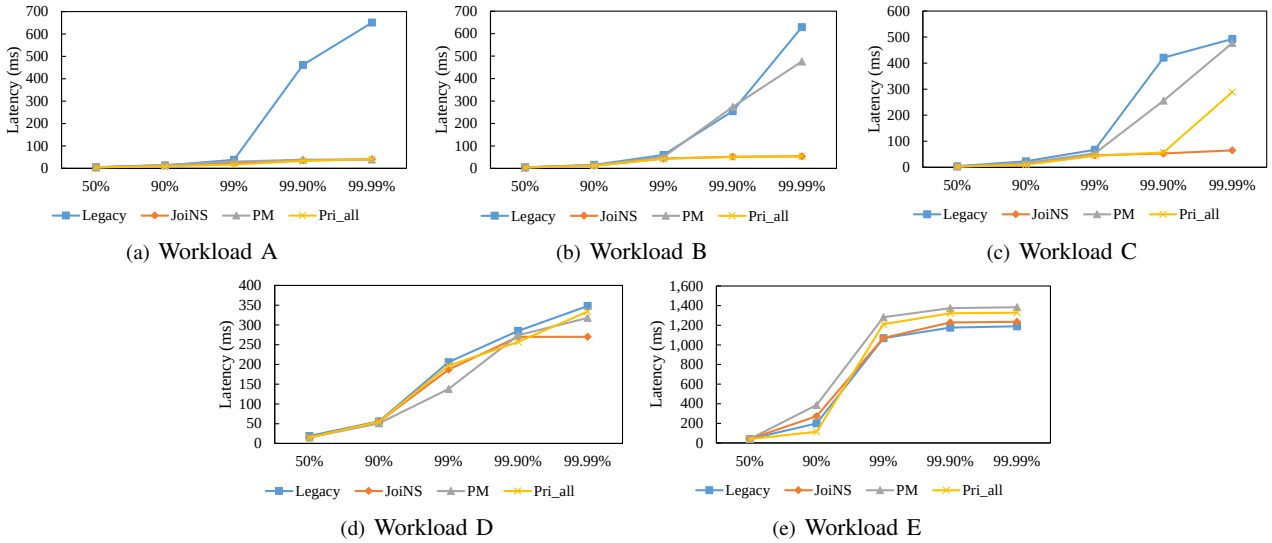
TABLE I: Workload traces used in our evaluation.

Workload label	Workload Description	Interarrival Variance
Workload A	MSR web staging stg_1	86.68
Workload B	MSR research projects rsrch_0	12.17
Workload C	MSR user home directories usr_0	6.55
Workload D	MSR print server prn_0	12.76
Workload E	MSR firewall/web proxy prxy_0	22.85
Workload F	Synthetic 70MB/s read I/Os, 30MB/s write I/Os	

B. JoiNS latency performance

In this section, we demonstrate the ability of JoiNS in meeting latency SLOs in a networked storage environment. In this experiment, we run workloads A-E at the same time. Each workload represents a client. All the workloads share network and storage resources. We replay traces based on timestamps for one hour. Meanwhile, we generate competing network traffic and storage traffic (as noise). By analyzing the throughput of each workload, we are able to generate proper noise so that the aggregated load transmitted on network and storage are around 95% of the total bandwidth, which is close to congestion. According to the load on network and storage in nature, we set a trace a higher latency SLO if it has a higher load. In this experiment, we set latency SLOs of workload A-E to be 35ms, 40ms, 45ms, 50ms and 55ms respectively. In JoiNS, the priority of each I/O is set at real time based on the system status and latency SLO. In PM, the priority ordering is selected as A,B,C,D,E from high to low.

Figure 2 shows the tail latency for these workloads by applying different policies. It is essentially a representation of CDF at different percentage of latency. The results are grouped by the workload and it is easy to see JoiNS outperforms other policies for all workloads. From Pri_all we see that simply prioritizing all read requests and write responses of a workload without the control of JoiNS does not perform as well as JoiNS. This is because those I/Os which deem the system as not congested or fully congested are also prioritized. Controlling these I/Os unnecessarily will bring additional computation overhead in network and storage nodes and also occupy more resources which should have been allocated to other I/Os. Comparing with PM, JoiNS only prioritizes those



(a) Workload A (b) Workload B (c) Workload C

(d) Workload D (e) Workload E

Fig. 2: Request latency of workloads running at the same time at different percentiles.

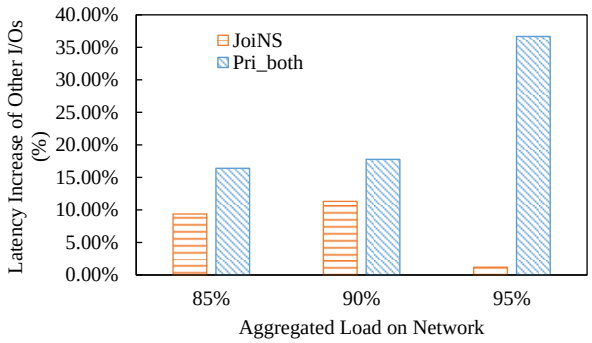


Fig. 3: The cost of prioritizing based on the asymmetry in read and write I/O packet size compared with prioritizing both the request and the response of an I/O.

I/Os whose estimated latency falls between βD and D. But in PM, once a priority of a workload is determined, all I/Os will go through the queue with that particular priority in network and storage regardless of network and storage status. Further, JoiNS only prioritizes read requests and write responses to reduce overhead to other I/Os while PM will also control read responses and write requests.

If we define that a workload is bursty when its squared relative standard deviation of the interarrival time is greater than 20, we have workload A,E as bursty workloads and B,C,D as not bursty workloads. We can see our JoiNS outperforms other policies regardless of burstiness.

C. Prioritization Cost

Our control is based on the asymmetry property in read and write I/O packet size, and we only prioritize read requests and write responses to generate a small overhead. In this experiment, we evaluate the cost of this approach. When we prioritize some packets, other packets will be delayed and their latency is sure to increase. The increase of latency of other packets is defined as prioritization cost. We use workload F and control read I/Os while leaving write I/Os for observation. We compare the percentage of latency increase of write I/Os in JoiNS with Pri_both. Figure 3 shows the cost when the network is close to congestion. It is obvious that the cost of JoiNS is much less than Pri_both.

VII. Conclusions

This paper present JoiNS, a system that coordinates different components along the I/O path to ensure latency SLO in an environment where clients access remote storage through network. JoiNS includes client, network and storage into control. It deploys a logically centralized controller to coordinate the control on each component. The controller has a global visibility of network and storage. It monitors the status of network and storage, and estimates the time needed on network and storage respectively for each I/O request. It determines whether to control I/Os along the path based on the SLO deadline, network and storage status, time estimation and I/O characteristics. Enforcers along the I/O path will adjust the priorities of I/Os accordingly. JoiNS is adaptive to the system status change and will only exercise control when there is a need. Through experiments, we show JoiNS is able to improve SLO compliance significantly.

In the future, we consider to apply the methods in sRoute [23] and reroutes I/Os when there is congestion. We will explore how to integrate this method into JoiNS to make it adaptive to the system status change. We will also explore more methods of control on storage in OpenStack [21], [6] and other cloud storage environment.

Acknowledgment: This work was partially supported by NSF awards 1305237, 1421913, 1439622, 1525617 and 1812537.

References

[1] Amazon s3. https://aws.amazon.com/s3/.
[2] Apple icloud. https://www.apple.com/icloud/.
[3] Google drive. https://www.google.com/drive/.
[4] Microsoft onedrive. https://onedrive.live.com/about/en-us/.

[5] Open vswitch. http://openvswitch.org/. Accessed: 2017-9-25.

[6] Openstack. https://www.openstack.org/. Accessed: 2018-04-16.

[7] Z. Cao, H. Wen, F. Wu, and D. H. Du. Alacc: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 309–323. USENIX Association, 2018.

[8] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, pages 1–12. ACM, 2007.

[9] M. Corporation. Server message block (smb) protocol. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-SMB/[MS-SMB].pdf. Release: June 1, 2017.

[10] N. Feamster, J. Rexford, and E. Zegura. The road to sdn. *Queue*, 11(12):20, 2013.

[11] S. Floyd. Tcp and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[12] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN)*, 1(4):397–413, 1993.

[13] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu. Openanfv: Accelerating network function virtualization with a consolidated framework in openstack. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 353–354. ACM, 2014.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[15] Y. Hua, X. Liu, and D. Feng. Smart in-network deduplication for storage-aware sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):509–510, 2013.

[16] S. McDaniel, S. Herbein, and M. Taufer. A two-tiered approach to i/o quality of service in docker containers. In *2015 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 490–491. IEEE, 2015.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[18] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[19] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[20] Seagate. Scsi commands reference manual. https://www.seagate.com/files/staticfiles/support/docs/manual/Interface%20manuals/100293068j.pdf. Release: October, 2016.

[21] O. Sefraoui, M. Aissaoui, and M. Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[22] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *FAST*, volume 12, pages 1–14, 2012.

[23] I. A. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sroute: Treating the storage stack like a network. In *FAST*, pages 197–212, 2016.

[24] A. Tanenbaum. *Computer Networks*, chapter 1, pages 161–164. Prentice Hall Professional Technical Reference, 2011.

[25] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196, 2013.

[26] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling high-level slos on shared storage systems. In *ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[27] H. Wen, D. H. Du, M. Shetti, D. Voigt, and S. Li. Guaranteed bang for the buck: Modeling vdi applications with guaranteed quality of service. In *Parallel Processing (ICPP), 2016 45th International Conference on*, pages 426–431. IEEE, 2016.

[28] Z. Wu, C. Yu, and H. V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, pages 543–557, 2015.

[29] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.