Adaptive Data Center Network Traffic Management for Distributed High Speed Storage

Madhurima Ray, Joyanta Biswas, Amitangshu Pal, Krishna Kant
Temple University, Philadelphia, PA 19122, USA

***Abstract*—The emerging high-speed storage technologies and the ever-growing need for data storage are placing increasing pressure on data center networks. In this paper, we develop novel mechanisms for dynamically deciding when to move storage chunks or alter the number of active copies to alleviate congestion during high traffic episodes and to enable traffic consolidation (and hence network energy savings) during low traffic periods. Both actions are essential due to increasing burstiness of data center traffic. Using extensive simulations with modified NS3 network simulator we provide deep insights into the migration vs. replication tradeoff. We also show that under traffic bursts, a careful replication can provide up to 47% improvement in latency with read-dominated traffic, and strategic data migration to a low utilized node can provide up to 22% improvement in delays with write dominated traffic.**

***Index Terms*—congestion control; FAT tree; replication; incremental optimization; data center network; distributed storage;**

I. Introduction

Storage technologies and protocols in data centers are undergoing a rapid transformation have already shattered the conventional wisdom that the networks are fast and the storage is slow [1]. For example, currently available high-end NVMe SSDs (e.g. Samsung MZPLL12THMLA) can put out up to 50 Gb/s for large sequential accesses such as streaming media [2]. Thus, a few SSDs in a storage server can overwhelm 100 Gb/s Ethernet. Faster technologies such as Intel Optane (*https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html*) not only consumes higher bandwidth, but also compete with end to end network latencies in larger networks.

Fortunately, the average IO or network traffic in a data center network is invariably very low and the real issue is not sustained high traffic but frequent bursts in traffic due to large data ingestion or output. We see this behavior even in our university cluster traffic. Fig. 1 shows the traffic pattern of one logical unit (LUN) over 11 hours of a day which has several sharp peaks. Thus, a well crafted dynamic migration (or move) and replication (or copy) of data can deal with such bursts and minimize QoS violations. At the same time, consolidation of traffic during low periods is necessary to (a) avoid additional latency because of (unintended) scattering of data throughout the network as copies are created, migrated, or removed, and (b) effectively exploit the low power mode of the links. Unfortunately, traffic consolidation and congestion mitigation are conflicting goals thereby requiring careful data placement and movement techniques.

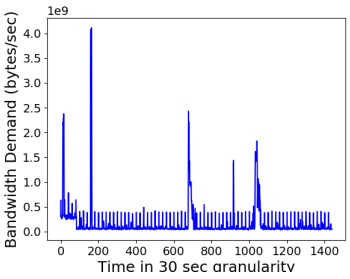

Fig. 1. Access time-series based on Temple University cluster data traces

Achieving this tradeoff involves several challenges. First, if packet drops are to be avoided, an ongoing flow cannot use a different copy of the data chunk, instead, any migration is limited to only new flows. Second, the migration/copy creation must be very lightweight so that the additional traffic generated does not worsen the congestion. Third, the additional overhead of consistency management across chunk copies in relieving network congestion should not affect the overall performance. Fourth, the replication/migration used during congestion episodes should not result in substantial scattering of data throughout the network as it worsens low traffic performance and ability to do energy management.

To address all these challenges, we have developed an incremental chunk management mechanism that adaptively combines chunk migration and replication, which is the main contribution of this paper. Extensive simulation shows that the combined chunk migration and replication mechanism provides 4x better performance than using them in isolation. For the simulation, we modify the existing NS3 simulator and show that in a disaggregated NVMe storage system under traffic bursts, our mechanism can provide up to 47% improvement in latency with read-dominated traffic, and strategic data migration can provide up to 22% improvement in delays with write dominated traffic as compared to an optimized initial placement without any further changes. To the best of our knowledge, no work has addressed above mentioned challenges in achieving congestion vs. consolidation tradeoff in disaggregated NVMe based storage.

The rest of the paper is organized as follows. Section II explores the background and motivation behind the work. Sections III and IV discuss the related work and the proposed chunk management methodology respectively. Section V discusses the optimal chunk placement and movement mechanisms, including both initial optimal placement and its incremental optimization. The experimental setup and results

This research was supported by NSF grants CNS-1422921 and IIP-1439672

are covered in Section VI and the paper is concluded in Section VII.

II. Motivation and Background

Storage systems are evolving rapidly from the slow spinning magnetic media to high-speed flash technologies (i.e., SSDs) and even higher speed technologies NVM technologies (e.g., PCM, MRAM, etc.), which can rival DRAM access latencies. Alongside there have been rapid developments in storage access protocols that are much leaner and lower in latency (e.g., the NVMe protocol that is rapidly becoming ubiquitous). This emerging high-speed storage will invariably be accessed over the data center network by many hosts regardless of how it is deployed (from fully centralized to fully distributed per server storage). With distributed applications deployed in VMs/containers on different hosts and accessing large amounts of data, remote storage access is a norm rather than an exception.

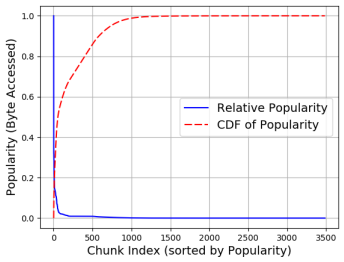

Fig. 2. Chunk Popularity based on Temple University cluster data traces

The net result of these trends is that storage systems can drive tremendous throughputs, although this typically happens only for brief periods. Furthermore, with storage device latencies going into 10's of microseconds or less, the network latency becomes a significant part of the end to end latency and needs to be managed carefully, especially during congestion periods. Thus an intelligent management of network congestion during burst periods becomes important. Since the congestion management generally requires spreading out the data (to spread out the network traffic), we also need to consider the fact that scattered data becomes undesirable during low traffic periods (which are dominant) as it interferes with the ability to achieve low network latency and exploit power-saving techniques (e.g., sleep modes) for links with low utilization. Thus, an intelligent mechanism needs to manage data location (by copying or moving) in a way to both avoid the congestion and to avoid keeping it scattered throughout the network.

The dynamic data placement to achieve the congestion mitigation vs. traffic consolidation tradeoff requires that the storage be divided up into "chunks" of a suitable size and virtualized so that it is possible to move these chunks dynamically without any impact on the applications in terms of addressing or accessing the data they need. Storage virtualization is a very mature technology and is used extensively. It may be performed by the host, a virtualization appliance, or the switch. While each mechanism has its pros and cons in terms of network impact and delays, we do not delve into those details here.

Despite virtualization, chunk movement without allowing ongoing transactions to complete can lead to lost or out of order network packets. This mobility limitation tends to make the problem of data movement/copy management more challenging, as discussed later in the paper. It is also worth noting that the storage access pattern is generally highly skewed, which means that a small fraction of chunks will account for a large percentage of accesses. This is illustrated in Fig. 2 for our university traffic by plotting the distribution of chunk popularity. It turns out that approximately 11% of the LBA (logical block addresses) are responsible for 80% of the traffic. In fact, the popularity distribution often turns out to be similar to Zipf. Obviously, only the highly popular chunks need to be moved or copied to deal with congestion; however, since highly popular chunks are likely to be accessed from multiple hosts, managing their mobility becomes quite challenging.

In addition to the data movement and copying explored here, dynamic per-flow changes to network routing can also be used to mitigate congestion and to consolidate traffic. We have studied this aspect in our earlier work [3]; therefore, we do not address it here. It is possible to integrate data movement/copying with intelligent routing, but this is beyond the scope of this paper. Also, while one of the objectives of traffic consolidation during low traffic periods is better network energy management, this paper is not focused on that aspect either. The copy management discussed in this paper could make use of the copies normally created for better resilience; however, our copying is limited only to very hot chunks that would likely reside in the highest storage tier and extra copies are likely to be short-lived.

III. Related Work

Several works focus on performance aware data placement and data replication from different viewpoints. In particular, references [4], [5] concentrate on placement in a cache tiering environment to achieve performance in terms of in-storage access latency, but the *network bottleneck* issue is overlooked. References [6], [7] focus on a distributed file system and database respectively and propose replication with *static workload estimate*. However, the authors do not consider the overhead due to *consistency control* inside the workload estimates.

The authors in [8]–[14] focus on the locality-aware data placement in a distributed storage system. References [8]–[10] mainly consider distributed big data applications (e.g, Hadoop and distributed database). In [8], the author uses the correlation between the number of accesses and concurrent accesses to find popularity. In [9] the author works on the same basis (chunk's popularity), but the popularity is determined by correlation analysis between access frequency and age of the file. The work in [10] additionally considers the host's capability for hosting the replicas while making the placement decisions. References [11]–[14] discuss network-aware end-point consolidation, with references [13] and [14] place VMs with more mutual communication close to one another to reduce delay. In [11], authors have extended the work of [15]

by considering VM utilization and correlation analysis for both the VMs and flows in a coordinated manner. Scheduling the data access can also improve the performance [16], but in a large distributed storage system this would introduce *additional overhead* to manage the coordination.

Locality aware data replication/placement has also been studied in Wide Area Network [17], [18]. GlobeDB [17] uses clustering of data chunks based on the read and write amount conducted by each server and place each chunk cluster to any single server based on a cost function, which considers three metrices: read, write and consistency traffic. Reference [18] studies different heuristics to solve the NP-hard chunk placement problem.

Contrary to the above approaches, in this paper, we propose (a) a dynamic data-chunk placement scheme that balances both congestion and consolidation depending on change in traffic bursts, (b) adapts to the traffic behavior (read/write) by using either replication or migration to mitigate congestion; (c) incrementally reacts to any perturbation in traffic; and (d) finally addresses all the overheads due to replication, migration, consistency control.

IV. Proposed Methodology

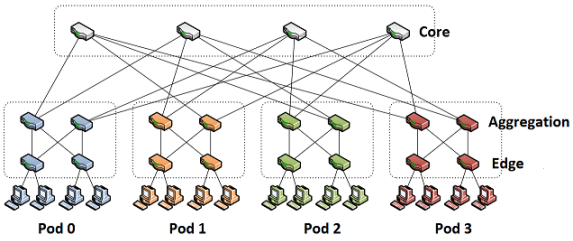

Fig. 3. An illustration of fat-tree network

We assume a traditional data center network architecture organized as a "fat tree" illustrated in Fig. 3. An order k fat-tree has k pods, each pod contains k switches arranged in two layers (aggregation and access) with $k/2$ switches each. The figure shows a rather small fat-tree of order $k = 4$. The leaf nodes connect to top-rack (ToR) switches, and each rack contains some number of compute and storage servers. All links are bi-directional and generally have higher bandwidth at higher levels of the hierarchy. Nevertheless, higher level links are more likely to be congested if the applications and the storage used by them are scattered throughout the network.

We assume that all storage is virtualized with the chunk size of 256KB. Since this paper is only concerned with storage traffic, we assume separate logical channels carrying the storage traffic. Such channels can be realized using MPLS at IP level or through the use of CoS (class of service) feature in data center Ethernet.

Although any storage chunk can reside on any storage server, it is desirable to keep the chunks close to the applications that access them the most while still consolidating the traffic under normal conditions. This can be done initially by solving an optimization problem, which we discuss in section V. Solving such an optimization problem requires knowledge of various access types and intensities, which may be available based on historical behavior, however, the access patterns are likely to change over time. Thus in addition to the initial placement, we also need a mechanism for continuous monitoring of the entire network and occasional incremental optimization that moves or changes the copies of a certain number of chunks. We expect the need for incremental optimization to arise only occasionally and likely involve only a small fraction of all the chunks.

Our monitoring architecture is similar to one proposed in [19] and involves a local controller (LC) residing at each switch and a global controller (GC) to coordinate among the LCs. The GC can run on the centralized management box, which usually has a logically separate communication path to all the switches. The well known SDN architecture provides a suitable paradigm for this [20], where GC can be thought of as a controller interacting with individual switches with additional monitoring capabilities.

The prime job of an LC is to collect utilization of its switch and that of the incoming/outgoing links periodically, and it can do this without much overhead. A GC occasionally communicates with the LCs in the system to build the holistic picture of the network nodes and links. The LCs at ToR switches (rack level) track the utilization of all the chunks hosted by that rack. For scalability, the LCs hide the server level details inside the rack and only expose a consolidated view of the endpoints externally to the GC. The GC acts upon the overall rack-level statistics whereas the LC controls any decision internal to the rack which is essential to keep the GC overhead under check.

Given this architecture, it is possible to make decisions about when a chunk should have a copy made, moved, or a copy deactivated. The overhead of such dynamic replication includes (a) additional traffic at the time of replication, and (b) the synchronization traffic at the time of any updates. Other than the GC and the LC, the Virtualization is responsible for all the mapping and the translation, where the mapping can be influenced by the GC or the LCs. But we would not discuss the API related details here.

To address these, we assume a mechanism that minimizes replication traffic during high traffic episodes by using either a *reactive* or a *proactive* approach based on the dominant traffic type as learned from the history of active chunks. The *reactive* approach determines the number of extra copies needed for heavily used chunks during traffic bursts and creates those copies inline. In contrast, the *proactive* approach creates, in advance of the traffic burst and based on historical data, a small number of copies of the chunks that are known to be hit heavily. These copies are retained during low traffic periods but not in fully synchronized state. Instead, they are synchronized opportunistically in the background using *lazy synchronization*. The key advantage of the proactive scheme is substantially less copy synchronization traffic during burst

periods; however, this comes at the cost of higher storage consumption. With either scheme, once the burst is gone, the redundant copies are deactivated.

V. Optimal Chunk Placement and Movement

A. Initial Placement of Chunks

We assume that the applications and the chunks used by them are placed optimally initially to minimize network traffic and adjusted incrementally as the traffic or applications change.

For initial placement, we formulate an optimization problem where x_{un} is a decision variable which is 1 if either the application or the chunk u are assigned to node n and 0 otherwise. Suppose r_i and r_o are the cost of accessing a chunk that is within and outside the node respectively, i.e. $r_o > r_i$. To linearize the problem, we also introduce an auxiliary decision variable e_{uv}, which is 1 if and only if the application u and chunk v are placed on different nodes. Let w_{uv} denote the weight (or relative intensity) of accessing chunk-v from application-u and I_u the IO demand by the application or chunk u. We could then define our *Traffic-aware Application-Chunk Placement (TACP)* as follows:

$$\text{Min} \sum_{(u\in\mathcal{A}\wedge v\in\mathcal{C})} w_{uv}e_{uv}r_o + w_{uv}(1-e_{uv})r_i \quad \text{s.t.} \tag{1}$$

$$\sum_{n=1}^{\mathcal{N}} x_{un} = 1 \;\; \forall u \in \mathcal{A}\vee\mathcal{C} \tag{2}$$

$$\sum_{u\in(\mathcal{A}\vee\mathcal{C})} I_u x_{un} \leq \mathcal{L} \;\; \forall n \tag{3}$$

$$w_{uv} = 0 \;\; \{u,v\} \in \mathcal{A}\vee\{u,v\} \in \mathcal{C} \tag{4}$$

$$e_{uv} \geq x_{un} - x_{vn}, \;\; e_{uv} \geq x_{vn} - x_{un}, \tag{5}$$

$$e_{uv} \geq x_{un} + x_{vn} - 1 \;\forall u, \forall v, \forall n \tag{6}$$

$$e_{uv} \in \{0,1\}, \;\; x_{uv} \in \{0,1\} \;\; \forall u, \forall v \tag{7}$$

Here constraint in eqn(2) states that every application and chunk copy is assigned to some node. Equation(3) states that the overall IO rate of the node (denoted $\mathcal{L}$) is respected. The constraint in eqn(4) states that there is no IO between two applications or two data chunks. Constraints in (5)-(6) ensure that if e_{uv} = 1, then u and v are placed on different nodes.

The above formulation can be used at multiple levels in a real network where a *node* could represent pod in entire fat-tree, rack within a pod, and servers within a rack. During the initial placement problem, the TACP is solved twice to determine in which *pods* and then which *nodes* inside that pod the application/chunks are assigned.

Theorem 1: The problem TACP is NP-hard.

Proof: This can be proved easily by a reduction from the Minimum K-cut problem (MKP) in a graph representing flows between nodes. We skip the detailed proof for brevity. ■

Because of the NP-hard nature of the TACP problem, we propose the following heuristics to solve it efficiently. We solve the problem in two stages. First, we assign apps/chunk-copies to a set of virtual nodes, each of which corresponds to a real node and has a specified IO capacity (i.e., every node is assumed to have the same IO capacity). Next, we map the virtual nodes to the real nodes of the network.

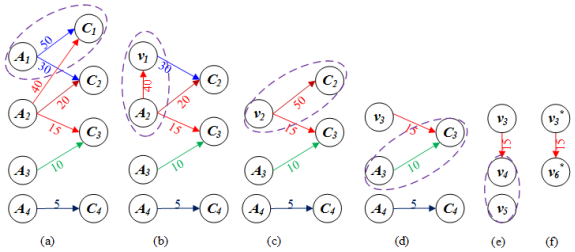

Fig. 4. Assigning apps/chunk-copies to the virtual nodes

In the first step, we construct an affinity graph Fig. 4(a) with application and data nodes (A_i's and C_j's) and weighted edges showing traffic demand from A_i's to C_j's. We then greedily choose the edge with the highest weight (i.e. $A_1 \rightarrow C_1$ in Fig. 4(a)) and merge its vertices to create a new composite vertex, and direct all incoming and outgoing vertices from/to the merged node to the composite vertex. Fig. 4(b) shows this with composite vertex v_1. The composite node corresponds to application A_1 and chunk C_1 being placed on the same node. The composite vertex is attempted to be merged further so long as the co-location does not exceed the capacity of a virtual node. E.g., in Fig. 4(d) merging v_3 and C_3 would exceed the virtual node capacity, and thus at this point, the algorithm looks to a distinct merger (to create another virtual node). The end result is two virtual nodes v_3 and v_4 that cannot be merged further.

At this point, there may be some disconnected components that are merged/packed into at most N virtual nodes using the typical bin-packing solution [21]. Thus v_3^* and v_6^* become two virtual nodes after the first step.

In the second step, the virtual nodes are assigned to the pods. For this, we start with the n virtual nodes from step 1, and if $n < N$, expand them to N virtual nodes by inserting $N-n$ dummy nodes. We then recursively apply Kernighan-Lin (K-L) graph bi-partitioning algorithm that minimizes the cut weights across two partitions. The K-L algorithm divides these N v-nodes into two partitions each one with a size of $N/2$ v-nodes (assuming N is even). The K-L algorithm is again applied to these two partitions if they have non-zero virtual nodes. This process is repeated until each partition has more k virtual nodes. These partitions can now be thought of individual pods, where the virtual nodes are assigned from left-to-right based on their app/chunk loads (from maximum to minimum).

Theorem 2: Assuming N is even, the K-L algorithm will be called at most $\left(\frac{N}{k}-1\right)$ times.

Proof: The K-L algorithm divides the virtual nodes into two equal-size partitions. Thus if we start with N virtual nodes, then from the formulas of G.P. series it is easy to verify that after $\left(\frac{N}{k}-1\right)$ steps we will get the partitions with the size of k. ■

Henceforth the term **"node"** explicitly refers to a **"rack"**, since all our strategies beyond this point are based on the rack-level estimates.

B. Incremental Placement Optimization

Since the application behavior and hence the network traffic will vary over time, it is essential to monitor the traffic constantly and incrementally make adjustments to the position or number of copies of highly used chunks.[1]

To facilitate chunk migration/replication, we assume that we initially deploy k copies (e.g., $k = 3$) for every chunk across the storage servers. These copies include the initial number of copies needed in the initial optimization (the active copies) and a few extra copies (inactive copies). The additional copies, if any, are placed according to some fair distribution rule, for which we use a simple hash-based approach described below. The purpose of the extra copies is to allow for inexpensive copy activation when required by the high demand without creating substantial additional network traffic. All inactive copies are synchronized opportunistically (to minimize activation cost) but active copies are kept entirely consistent.

In addition, the GC maintain a bitmap $BM(c)$ for each chunk c where each entry is primarily k bits long (we use $k = 3$). Thus if $BM(c, i)$, $i \in 0..k-1$ is 1(0), then a predefined hash function $h_i(c)$ for this position provides the location for the ith active(inactive) copy of chunk c. The GC also stores the activation ordering of any chunk using $k-1$ additional bits which it uses during copy deactivation (thus each BM entry costs $2k-1$ bits for any k). The LCs do not require access to this bitmap. The mechanism will, however, require communication between the GC and the virtualization engine during the time of migration/replication, but detailed consideration of the overheads of this interaction is beyond the scope of this paper.

We assume that the episodes involving the arrival of large traffic bursts and their subsequent dissipation are infrequent, as observed from several traffic traces. Therefore, the frequency of update of BM is expected to be relatively low. Nevertheless, in very large data centers, the maintenance of a centralized data structure such as BM may be undesirable. This aspect will be explored in our future research.

1) Copy Activation and Deactivation: For both replication and migration of chunks, we use the following mechanism based on the traffic monitoring (in bytes/sec) by the LC. For replication, the LC watches the outgoing traffic of the node (this is primarily the read traffic), and for migration, the incoming traffic (this is primarily the write traffic). When this traffic overshoots a certain high cutoff threshold τ_h (shown as 75% in Table I), the LC reports the event to the GC along with the top P highly used chunk numbers with their respective utilizations.

[1]It is also possible to occasionally re-run the initial optimization but this may not be desirable if it leads to substantial data movement.

Suppose that the current utilization of the i-th chunk is U_i. Then we have the following situations:

Replication: Suppose that chunk i has n active replicas with evenly distributed load among them. Then, creating another copy (replica) of that chunk will reduce their estimated load to approximately $\frac{n}{n+1}U_i$.

Migration: Migrating the chunk from node m to p will reduce load U_i from node m and increase U_i at destination node p. For any chunk with multiple active copies in the system, the incremental optimization migrates only the copy from the node that has raised the event, while keeping the locations of the remaining active copies being unchanged.

Based on this approximation the GC selects chunk from the list in descending order of their utilization at the nodes and activates the replicas at some other node that (a) has an inactive replica of that chunk, (b) can accept a certain amount of additional load (i.e. $\frac{n}{n+1}U_i$ in case if replicating the i-th chunk) and (c) has the least load (preferably preoccupied) satisfying (a) and (b). The GC continues to activate replicas until the expected load of the congested node goes below the normal limit of τ_{n2}. In case there is no node that satisfies these conditions, that chunk is not replicated or migrated and GC moves to the next chunk in the sorted list.

When the traffic surge fades away, our scheme needs to deactivate some copies activated during congestion so that the overhead of maintaining the consistency among the replicas is avoided. A migration also involves a copy creation initially followed with new traffic directed to the copy. Only when the ongoing traffic dies down, the original copy is deactivated. Deactivating extra copies when the traffic subsides can be rather tricky. Deactivating it too early after the traffic burst subsides could hurt in terms of delay since there still might a built-up backlog that must be cleared.

There could be several policies for copy deactivation. Each of them has its advantages and downsides. It may be difficult to find a single copy with low utilization under our traffic forwarding mechanism. So infrequently, the LC updates the GC with the set of Q chunks having low utilization. From the list, GC selects only the chunks with multiple active copies. It then deactivates the last created copy of the chunk and shifts its load equally to other active copies. (Note that no deactivation will happen if the remaining copies cannot handle the extra load).

2) Concurrency Control: Since we use multiple copies as a way of relieving congestion, we must address the multi-copy consistency issues as well for chunks that are shared across applications and thus could get access requests for them asynchronously from multiple applications. As usual, this can be achieved in two ways: (a) via locking or pessimistic concurrency control (PCC) or (b) via optimistic concurrency control (OCC) which rolls back any conflicting transactions [22]. It is well known that the OCC works better in environments with low contention for the resources. Assuming that the chunk sharing is not prevalent, the contention is expected to be

low, and hence we choose to implement the OCC. In OCC, the resource access by a transaction must still be monitored; however, the requester can proceed without locking. When the transaction is ready to commit, a check is made if any other transactions have committed since this transaction started. If so, the transaction is rolled back and may retry after some delay to avoid the conflict. The transaction also checks for conflict upon arrival and backs off if another transaction has already started. Conflicts are still possible since there is a small gap between determining that there is no conflict and actually recording that the transaction has started.

It is important to note that our concurrency control operates at the storage level and is only concerned with individual read and write operations. Additional concurrency control may be applied at a higher level (e.g., for database transactions running on top of the storage system).

VI. Experimental Evaluation

A. Experimental Setup

To comprehensively evaluate the network congestion control mechanism we have enhanced the popular NS3 network simulation package to include the modeling of chunk based storage access, concurrency control, and traffic control via local controllers (LC) and global controller (GC). We built a small fat tree infrastructure for $k = 4$ with 100 Gb/s network links in the core and the aggregate layer and 10 Gb/s at the access or edge level.

For the storage, we consider device characteristics similar to those for currently available low-latency NVMe SSDs. However, we do not consider or simulate the complexities of the storage systems here – assuming that the storage device utilization is kept reasonably low, the effect of queuing for storage devices is not important to our study. For routing, we use the Equal-Cost Multi-Path Routing (ECMP) with a point to point connections. Also to avoid acting upon intermittent short-lived spikes, we perform *exponential smoothing* of the congestion metric. Some of the key parameters for the experimental setup are given in Table I.

TABLE I
Simulation Configuration Values

Parameters	Values	Parameters	Values
Fat Tree Size (k)	4	Mid Cut-off (τ_{n2})	55%
Zipf dist.(α)	0.8	Normal Cut-off (τ_{n1})	45%
#application instances	47	Default chunk size	256K
#unique data chunks	1500	SSD Read Latency	$25\mu s$
#servers per rack	14	SSD Write Latency	$100\mu s$
High Cut-off (τ_h)	75%	Smoothing factor (β)	0.85

1) Application and Data Chunk Model: In our system, we have M application instances and N data chunks. We define the data chunk access intensities following a Zipf distribution over the N ($N > M$) unique chunks with the decay factor $\alpha = 0.8$. We then randomly map the applications to the chunks. Based on the mapping between an application to the set of chunks, we determine the application frequencies. Without any sharing on the chunks across applications, the application set should also follow a Zipf distribution. Additionally, we share each chunk across a small set of applications.

2) Copy Activation Overhead: As stated earlier we can either create copies proactively or reactively. Under *proactive control*, we decide m alternative locations per chunk using m hash functions and create copies in advance, which are then synchronized in the background. In *reactive control*, we create copies as needed (i.e., at the time of congestion). Additionally, under replication, as we keep multiple copies of chunks active and share them across a small number of applications, we use Optimistic Concurrency Control (OCC) model to keep the copies consistent.

3) Chunk size and Metadata Overhead of our Scheme: The suitable chunk size is configured by the virtualization layer and it involves balancing the overhead of metadata maintenance (which prefers large sizes) vs. the overhead of moving chunks and controlling false data sharing (which prefers small chunks). For our mechanism, smaller chunks are preferred, and the chosen size of 256KB is unlikely to be burdensome in most situations because of the highly skewed nature of storage accesses. Note that our mechanism only needs to store the chunk id, offset, and three active locations of that chunk which results in a few bytes (32 bytes) per chunk. This has a miniscule storage overhead (0.0125%).

As described in section V-B, for the bitmap GC uses only 5 bits per chunk (for 3 extra copies), hence for a petabyte of storage with 256K chunk size, the bitmap will require 2GB of memory space. GC also stores the node level utilization for all the nodes to decide the best-fit location for the chunk while replication or migration. LCs do keep track of the chunks inside a single node which constitutes of chunk number, server number, and utilization. We make the assumption that at most 10% of the data is active at any point in time for which LC requires an in-memory monitoring structure. Given the chunk usage are roughly balanced across the pods in the data center (but there could be local imbalance inside a pod) each LC will require 0.5GB to 1.5GB of memory for monitoring.

4) Application Traffic Generation Model: We have largely used synthetic traffic for our evaluation as it allows for an easy change in the traffic parameters. We generate traffic based on the statistical characteristics of real block device access traces collected from both Temple University Cluster[2] and [23]. Analysis of the trace file in section I shows several periods of high activity, which we term as traffic *"burst"*, and develop our traffic generation model around these traces.

In our model, network flow duration follows Pareto distribution with shape=3.5 and mean as 3.6 ms. The bandwidth of each flow follows a uniform distribution with mean 1Gbps and a range from 500 Mbps to 1.5Gbps. To avoid the remote communication latency we keep the bisection bandwidth of the network topology adequate. We measure performance as – *the average and the maximum observed latency faced by any*

[2]https://drive.google.com/file/d/1gSCToOCk4oigxLd8jKO3LRLlMWoblr9C

application issuing remote storage access request.

B. Experimental Results and Discussion

1) Optimized vs. Random Initial Placement: We start with a comparison of our optimized initial placement (*OIP*) against a uniform random placement (*URP*) of both the applications and the chunks. Under 20% utilization, the *OIP* only occupies 8 out of 16 available nodes (racks). In contrast, the *URP* involves all nodes as expected. We use the same traffic for both the cases and compare the performance of both the placements.

Fig. 5 shows the comparison of average latency under changing read fraction (95% to 1%). With bidirectional traffic, the network link gets congested at the extreme points where the traffic is either read or write dominated. At 95% read or 99% write traffic, we achieve the maximum reduction (approximately 95%) in average latency with *OIP*. Also *OIP* does not have any packet delivery issues whereas *URP* fails to deliver approximately 1% of the packets by the end of the simulation due to long delays. As long as the resources are available, *OIP* tries to consolidate the applications with their associated data chunks and accommodates them in the same pod (perhaps on the same node) to reduce packet propagation delay. Although mixed read/write traffic results in lower congestion due to bidirectional links, *OIP* can still achieve 18% reduction in average latency at 50% read traffic.

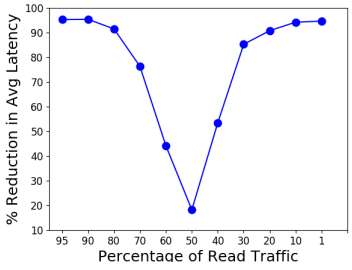

Fig. 5. Avg. Latency for Random vs optimized placement

The *OIP* will no longer be optimal under traffic bursts, and incremental optimization becomes necessary. It replicates or migrates busy chunks depending upon which direction of a bidirectional link is congested. Both operations require activation of additional copies of chunks at different nodes. In the case of replication, all future requests are evenly distributed among all active copies. In contrast, in the case of migration, all new requests go to the new copy, and when old requests die down, the old copy is deactivated.

2) Impact of Incremental Optimization: Figs. 6 shows the effect of varying percentage of read on average delay under 20% utilization. At that utilization, our *OIP* only uses half the number of nodes, which increases the utilization of those resources to higher than 20%. The rest of the nodes are not at all utilized. We assess the performance of incremental optimization by comparing average latency for the requests under two cases; in one, we only use the *OIP* (denoted as "no opt") whereas in the other we use incremental optimization over the *OIP* to change the number or location of active copies to accommodate the change in demand.

a) Change in average delay with changing read fraction: The following scenario can be treated as the best case for the incremental optimization where we create inactive copies *proactively* and perform *lazy synchronization*. We create inactive copies of working set on all the nodes. Hence during congestion, the traffic that we need to send to activate those inactive copies is mostly very small. Also, assuming copies everywhere (*"copies everywhere"*) gives us the flexibility to activate the replica on any node. Here in the current experiment independent of the ongoing traffic characteristics (read/write), we either only migrate or replicate.

Fig. 6(a) shows the average latency with and without the incremental optimization following the *OIP*. With read dominated traffic, the incremental optimization (which creates copies of the busy chunks) achieves approximately 47% improvement over *no opt*. This improvement results from the enhanced parallelism due to the extra copies. However, with write dominated traffic, extra copies cost additional synchronization traffic which actually increases the delay significantly as compared to *no opt*. Instead, a migration under write-heavy traffic achieves a 22% latency reduction. Obviously, migration is not helpful under read-dominated traffic. It follows that we need an adaptive mechanism that can track the congestion and the nature of the traffic (e.g., read fraction) and accordingly decides whether to copy or move chunks.

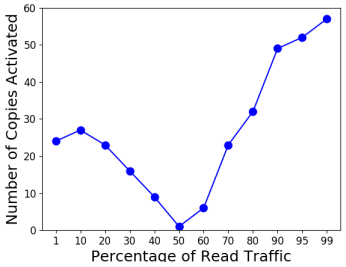

Fig. 7. No. of copies activated with adaptive control

So we introduce adaptive control based on the read fraction. The system decides whether we copy or move the busy chunks. Fig. 6(b) compares the average latency of this mechanism against *no opt*. When the traffic is mostly read; the system tries to reduce the congestion by replicating but when the traffic largely write dominated, the system reduces the consistency control overhead by migrating the busy chunks. Please note when there is not much traffic ongoing in either direction (read% 50-60%), neither copy nor migration is beneficial as the congestion in both the directions becomes insignificant.

b) Change in maximum observed latency with changing read fraction: Fig. 6(c) depicts the change in maximum observed latency that supports our findings from incremental optimization (Fig 6(a)). The maximum observed latency for any flow decreases by 14% compared to *no-opt* as we replicate due to parallel request service. However, when we migrate, the maximum observed latency remains identical to that for *no opt*. So by migrating, we do not lose anything in terms of maximum observed latency, whereas copy creation always reduces the maximum observed latency. Fig. 6(d) confirms our findings from Fig. 6(b) when tested under adaptive control and compared against *no opt*. It shows improvement in maximum observed latency is achieved through replication under read dominated traffic.

c) Number of copies activated vs. read fraction under different policies: Fig. 7 shows the number of additional

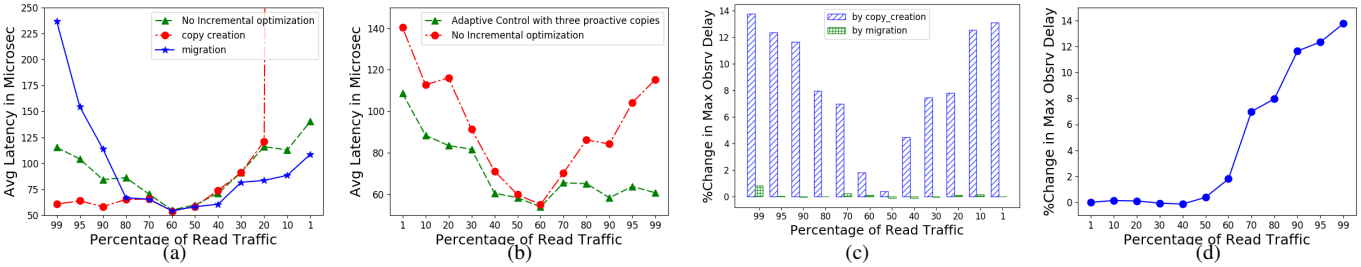

Fig. 6. (a,b) Average latency and (c,d) maximum observed latency reduction for various chunk management mechanisms

copies that are activated (and eventually deactivated with replication) over time. Across all the cases, in the worst case, we only activate a maximum of 50 additional copies out of all 1500 chunks. However, copy deactivation in our system follows a conservative approach. This sluggish deactivation might not have any impact on read dominated traffic, but it could increase concurrency control traffic when we replicate under write-heavy traffic.

Due to the performance advantage of the adaptive mechanism over simple migration/replication, we use the *adaptive mechanism* for the rest of our experiments.

From the perspective of efficient copying/migration, we would like to have a copy of each chunk available at each node. However, this would require an enormous amount of extra storage. Hence to keep the number of copies limited, we tested our system with three (*three copies*) and five (*five copies*) proactively created copies of all the chunks. We compare the performance of these cases against *copies everywhere*. Fig. 8(a) shows that *three copies* are enough to achieve significant performance improvement. With mostly read traffic, no significant performance improvement is observed beyond *three copies*. However, with write traffic, increasing the number of inactive copies can provide better location choices for migration, but at the cost of increased consistency control overhead.

Many-a-times the system fails to find a new destination for a busy chunk due to IO rate limitations at nodes hosting the inactive copies. Note that we determine the locations of the inactive copies in advance, and in a highly dynamic system, it is very likely that the action taken for one busy chunk can influence the migration/replication decision of another busy chunk. So if we now compare the number of inactive copies that are successfully activated during the bursts Fig. 8(b), fewer chunks are activated with the limited number of copies than for *copies everywhere*. For example with 90% read traffic, we activate 34 total copies with *three copies*, 35 with *five copies* and 49 with *copies everywhere*.

We do see a few exceptions; e.g., at 40% read, *three copies* creates two more copies than *copies everywhere*, yet the latency is 2% less in the latter case. This suggests that as the system fails to find a destination for a busy chunk, it chooses a less important chunk. Creating copies of such chunks requires more migrations/replications to bring the congestion down.

Since keeping *three copies* is enough to handle congestion,

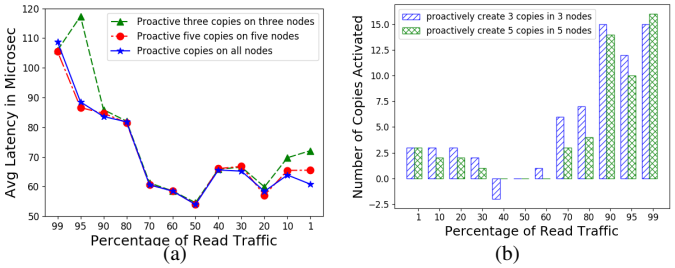

Fig. 8. Comparing (a) Average delay and (b) number of copy activated under limited copy approach over copy everywhere

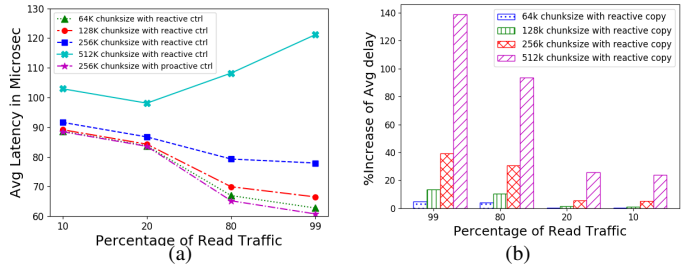

Fig. 9. (a) Average delay with proactive and reactive control and (b) Increase in delay during bursts over proactive approach

we use this configuration for further experiments.

3) Impact on Average delay under reactive copy creation with varying chunk size: Figs. 9 show the impact of increasing chunk size on average latency with inline copy creation (as opposed to our proactive method). In Fig. 9(a) we show the average latency by varying the chunk size from 64k to 512k. We compare the performance against our proactive approach with *three copies (default chunk size)*.

As stated above, our default chunk size is 256k. We perform the test with four different read proportions consisting of either read or write-heavy traffic. With read dominated traffic, the delay increases with the chunk size. At 99% read, compared to the proactive approach, the reactive method with a chunk size of 512k increases overall latency by 99%. This latency increase is due to the copy creation traffic that puts additional load on read path itself. For the same reason, with the write-heavy traffic, the increase in delay due to reactive copy creation decreases significantly. With 90% write traffic, we see an increase of 0.2%, 1%, 5%, 23% in delay with chunk size 64, 128, 256, and 512k respectively. Fig. 9(b) zooms into the bursts period, and the plot shows the increase in delay over proactive copy which confirms the same. Here due to bigger chunk size, the delay increases by 140% at 99% read fraction, which is more than the increase in overall delay.

Thus our findings from Fig. 8 and Figs. 9 indicate that with write-heavy traffic, one can afford to have reactive copies as long as the chunk size is small to moderate. With limited copies, activation may fail due to the shortage of resources at those nodes. Reactive copies will reduce the concurrency control traffic and might provide better locations for migration. In contrast, with read oriented traffic, it is beneficial to create copies proactively.

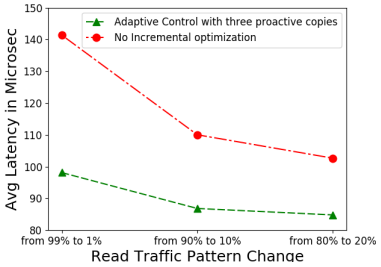

Fig. 10. Avg latency vs. read fraction

4) Adapting to Traffic type and Intensity Changes: Finally, we tested our system under not just change in traffic intensity, but traffic where both the intensity and the read-write ratio changes. Fig. 10 shows the change in average delay when both traffic type and intensity changes. We use three cases where traffic type changes from (1) 99% to 1% (2) 90% to 10% and (3) 80% to 20% read, with traffic intensity increases as before. We use *three copies* and compare the performance against *no opt*. Please note that in all our previous experiments, the read fraction was constant throughout a simulation and only varied across different runs. However, in this experiment, the read fraction of the traffic changes within a single run to test the adaptability. We found that our mechanism adapts nicely by both migrating and replicating chunks based on the change in traffic intensity and type. Also, we achieve 30% improvement over no optimization as we move from completely read to write heavy traffic.

VII. Conclusions and Future Work

In this paper, we discussed an adaptive mechanism to decide whether to migrate, activate an inactive copy, or reduce the number of active copies to handle changes in the network traffic due to variations in network traffic generated by high-speed storage devices that are becoming the norm in the data center. Through extensive simulations, we show that it is possible to achieve near 47% improvement in average delay with purely read traffic. However, for chunks that are mostly written, the best strategy is to migrate them out of the congested node and place them at some low utilized node.

In the future, we would like to study the scalability of the interactions between the GC, LC, and virtualization engine (VE) in a moderate to large-size data center environment. Assuming switch-based virtualization, the LC and VE functionalities can be integrated into a single module which avoids the overheads associated with separate entities. Furthermore, the GC can use a federated model whereby each pod has a lower-level GC that communicates with its peers to keep an overall network view consistent across all the GCs. However, this straight forward design still has many challenges that are to be addressed to achieve performance at scale, which we would like to explore.

In addition, we would like to integrate data migration/replication with intelligent routing through flow consolidation on network links to design a storage and network-aware consolidation vs congestion trade-off mechanism.

References

[1] R. Pydipaty, J. George, A. K. Saha, and D. Dutta, "The effect of non volatile memory on a distributed storage system," in *IEEE HiPCW*, 2018, pp. 11–17.

[2] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A tiered file system for non-volatile main memories and disks," in *17th USENIX (FAST 19)*, 2019.

[3] M. Murugan, D.Du, and K. Kant, "On the interconnect energy efficiency of high end computing systems," *SUSCOM*, April 2012.

[4] Z. Yang *et al.*, "Autotiering: Automatic data placement manager in multi-tier all-flash datacenter," in *2017 IEEE 36th (IPCCC)*, 2017, pp. 1–8.

[5] T. Wang, J. Wang, S. N. Nguyen, Z. Yang, N. Mi, and B. Sheng, "Ea2s2: An efficient application-aware storage system for big data processing in heterogeneous clusters," in *2017 26th (ICCCN)*, 2017, pp. 1–9.

[6] J. Xiong, J. Li, R. Tang, and Y. Hu, "Improving data availability for a cluster file system through replication," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[7] E. Zamanian, C. Binnig, and A. Salama, "Locality-aware partitioning in parallel database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 17–30.

[8] Ananthanarayanan *et al.*, "Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters," *EuroSys '11*, pp. 287–300, 2011.

[9] C. L. Abad, Y. Lu, and R. H. Campbell, "DARE: Adaptive data replication for efficient cluster scheduling," *IEEE ICCC*, pp. 159–168, 2011.

[10] Xie *et al.*, "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters," *IPDPSW 2010 IEEE International Symposium on*, vol. 9, pp. 29–42, 2010.

[11] K. Zheng, X. Wang, L. Li, and X. Wang, "Joint power optimization of data center network and servers with correlation analysis," in *IEEE INFOCOM*, 2014, pp. 2598–2606.

[12] H. Jin *et al.*, "Joint host-network optimization for energy-efficient data center networking," in *IEEE IPDPS*, 2013, pp. 623–634.

[13] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *IEEE INFOCOM*, 2010, pp. 1154–1162.

[14] D. Li, J. Wu, Z. Liu, and F. Zhang, "Joint power optimization through vm placement and flow scheduling in data centers," in *IEEE IPCCC*, 2014, pp. 1–8.

[15] X. Wang, Y. Yao, X. Wang, K. Lu, and Q. Cao, "Carpo: Correlation-aware power optimization in data center networks," in *IEEE INFOCOM*, 2012, pp. 1125–1133.

[16] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *ACM EuroSys*, 2010, pp. 265–278.

[17] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen, "Globedb: Autonomic data replication for web applications," in *Proceedings of the 14th International Conference on World Wide Web*, 2005.

[18] M. Karlsson and C. Karamanolis, "Choosing replica placement heuristics for wide-area systems," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, 2004.

[19] J. Biswas, M. Ray, S. Sondur, A. Pal, and K. Kant, "Coordinated power management in data center networks," *SUSCOM*, vol. 22, pp. 1 – 12, 2019.

[20] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. Van Reijendam, P. Weissmann, and N. McKeown, "Maturing of openflow and software-defined networking through deployments," *Computer Networks*, vol. 61, pp. 151–175, 2014.

[21] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[22] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981.

[23] A. Verma, R. Koller, L. Useche, and R. Rangaswami, "Srcmap: Energy proportional storage using dynamic consolidation," in *8th USENIX FAST*, Berkeley, CA, USA, 2010, pp. 20–20.