I/O Load Balancing for Big Data HPC Applications

Arnab K. Paul†, Arpit Goyal†, Feiyi Wang‡, Sarp Oral‡, Ali R. Butt†,
Michael J. Brim‡, Sangeetha B. Srinivasa†
†*Virginia Tech*, ‡*Oak Ridge National Laboratory*
{akpaul, arpitg, butta, bsangee}@vt.edu, {fwang2, oralhs, brimmj}@ornl.gov

***Abstract*—High Performance Computing (HPC) big data problems require efficient distributed storage systems. However, at scale, such storage systems often experience load imbalance and resource contention due to two factors: the bursty nature of scientific application I/O; and the complex I/O path that is without centralized arbitration and control. For example, the extant Lustre parallel file system—that supports many HPC centers—comprises numerous components connected via custom network topologies, and serves varying demands of a large number of users and applications. Consequently, some storage servers can be more loaded than others, which creates bottlenecks and reduces overall application I/O performance. Existing solutions typically focus on per application load balancing, and thus are not as effective given their lack of a global view of the system.**

In this paper, we propose a data-driven approach to load balance the I/O servers at scale, targeted at Lustre deployments. To this end, we design a global mapper on Lustre Metadata Server, which gathers runtime statistics from key storage components on the I/O path, and applies Markov chain modeling and a minimum-cost maximum-flow algorithm to decide where data should be placed. Evaluation using a realistic system simulator and a real setup shows that our approach yields better load balancing, which in turn can improve end-to-end performance.

I. Introduction

High performance computing (HPC) is routinely employed in many science domains such as Physics, and Geology, to simulate and understand the behavior of complex phenomena. Big data driven scientific simulations are resource intensive and require both computing and I/O capabilities at scale. There is a crucial need for revisiting the HPC I/O subsystem to better optimize for and manage the increased pressure on the underlying storage systems from big data processing.

Several factors affect the I/O performance of big data HPC applications. First, the number and kinds of applications that an HPC storage system supports is increasing rapidly [32], which leads to increased resource contention and creation of hot spots where some data or resources are consumed significantly more than others. Second, the underlying storage systems ,e.g., Ceph [29], GlusterFS [3], and Lustre [5], are often distributed, and adopt a hierarchical design comprising thousands of distributed components connected over complex network topologies. Managing and extracting peak performance from such resources is non-trivial. With changing application characteristics, static approaches (e.g., [6], [27]) are no longer sufficient, necessitating dynamic solutions. Third, the storage components can develop load imbalance across the I/O servers, which in turn impacts the performance and time to solution for the big data problem. Consequently, achieving load balancing in the storage system is a key for achieving a sustainable solution.

Load balancing for HPC storage systems is crucial and is being actively studied in recent works [7]. Extant systems typically attempt to perform load balancing by either having limited support for read shedding to redirect read requests to replicas of the primary copy, e.g., in Ceph [16], or performing data migration. Alternatively, per-application load balancing has also been considered to balance the load of an application across the various I/O servers [27]. These existing approaches lack a global view of all the components in the hierarchical structure of the system, and mainly focus on only a small subset of metrics (e.g., only the storage capacity, and not performance of the components). Thus, these approaches cannot guarantee that the aggregate I/O load of multiple big data applications concurrently executing atop a parallel file system (with bursty behavior) is evenly distributed.

Consider the Lustre file system that forms the backend storage in, among other HPC systems, Oak Ridge National Laboratory's (ORNL) Titan supercomputer [1], [8]. The default strategy in Lustre is to allocate storage targets to I/O requests using a round-robin approach. Experiments show that this approach is inclined to either under- or over-utilize the resources due to the bursty nature of applications.

In this paper, we address the load imbalance problem in Lustre by enabling a global view of the statistics of key components. We select Lustre to showcase our approach as Lustre is deployed on 60 of the top 100 fastest supercomputers [8], and improving its performance will benefit a wide range of applications and users. We go beyond just network load balancing, e.g., as in NRS [22], or per-application approaches, e.g., as in access frequency-based solutions [27], to ensure that the Lustre Object Storage Targets (OSTs) that actually store and serve the data along with other I/O system components are load balanced. We leverage the existing hierarchy of Lustre to avoid introducing additional performance bottlenecks, and colocate the global component of our load balancer on Lustre's Metadata Server (MDS) that has a global view of all other components.

Our goal is to improve the end-to-end performance of HPC storage systems for big data applications. Our data-driven approach learns system behavior to better manage the load across various Lustre components. Specifically, we make the following contributions.

- We design a model for the Lustre file system to incorporate a load balancing strategy that considers the global view of the system parameters.

- We utilize a scalable publisher-subscriber model to monitor and capture the load of key components in Lustre. We use a Markov chain model that learns and predicts the future behavior of the application using the monitored data, and a minimum-cost maximum-flow algorithm to assign storage targets in a global load aware fashion.
- We design a realistic trace-driven Lustre simulator that captures the load imbalance behavior. We use the simulator and a real setup to study our approach and design decisions therein.
- We also evaluate the effectiveness and scalability of our approach. Experiments show that our approach helps in achieving a better load-balanced storage servers, which in turn can yield improved end-to-end system performance.

II. Background and Motivation

In this section, we first present an overview of the existing load balancing used for Lustre. Then we present a quantitative study of the load imbalance in the HPC I/O subsystem to motivate our approach.

The default Lustre implementation uses a round-robin approach coupled with disk utilization measure to balance the load across Object Storage Targets (OSTs). The first available OST is selected to store a file and if the OST still has available space (more than a predefined threshold), is then placed at the end of a list for the next round . This technique does not consider the load on other resources (MDS or Object Storage Servers (OSS)) or the I/O load on OSTs, and can lead to performance degrading hot spots.

Network Request Scheduler (NRS) [22] aims to achieve distributed network load balancing at the Lustre server level by reordering incoming RPCs to a Lustre server (e.g., MDS or OSS) so that individual Metadata Storage Targets (MDTs) or OSTs running on the server receive a fair share of the server's network resources. Our approach differs from NRS in that we go beyond considering only the network resources to include the many factors affecting I/O performance on various Lustre components (Table I), and aim to provide a globally balanced I/O subsystem to offer more stable I/O performance.

A. I/O Performance Statistics

Lustre is a hierarchical system. We identify the factors that affect the I/O performance in every layer of the system as shown in Table I. We utilize the files `/proc/meminfo` and `/proc/loadavg` to capture the memory and CPU utilization, and also read the values of Lustre parameters in various components of the hierarchical system, e.g., via `obdfilter.*OST*.brw_stats`.

The performance for serving metadata and the I/O rate of serving the actual data to the clients is dependent on the network performance. A congested network affects the I/O performance adversely. The network bandwidth information can be extracted using the `lnet stat` interface (`/proc/sys/lnet/stats`), which runs over LNET and Lustre network drivers.

B. The Need for Load Balancing

We conducted a simulator-based study to demonstrate the need for balanced load placement across Lustre components. Our simulator (Section IV-A1) faithfully implements the functionalities of various components in Lustre. In our model, we have 8 OSSes and every OSS is linked to 4 OSTs (a total of 32 OSTs). We use 24-hour traces from the combination of three application traces to drive the simulator. Two of the application traces are from the Interleaved-Or-Random (IOR) benchmark [13] and the Hardware Accelerated Cosmology Code (HACC) Application I/O kernel [12], while the third application trace is generated from a HPC Transaction Processing Application (TPA) running at a large financial institution [26]. For this test, we use the default round-robin approach of Lustre for allocating OSTs for file creation requests. All results shown are the average taken from three runs.

We measured load balance by taking the ratio of maximum system load to the mean system load—system load for OSTs is the disk space used, and for OSSes is the CPU utilization. This ratio should be one for ideal load balance. For OSTs, we measure the ratio of maximum disk space consumed taking all OSTs into account to the mean disk space consumed for every hour for 24 hours as shown in Figure 1. We see that the system starts from a highly unbalanced setup of OSTs and takes over 12 hours to get close to 1. Thus round-robin is very slow in providing load balance.

In addition to load balancing OSTs, our objective is to also have a load balanced set of OSSes. We study the ratio of maximum CPU utilization taking all OSSes into account to the mean CPU utilization for a period of 24 hours with 1-second interval as shown in Figure 2. This ratio should also be ideally 1. As seen in the graph, there are huge fluctuations in the ratio for a very long window of 20 hours, again highlighting the weakness of the round-robin approach.

Since, load on OSTs is a continuous function (not represented in Figure 1), we also plot load of OSTs along time in a box-plot. Capacity in an OST is the amount of free disk space available. As the capacity continuously changes every hour, we normalize it to the median OST capacity for that hour. This is shown for the studied 24-hour period in Figure 3. The box plot highlights the variation in the capacities of all OSTs combined for different hours. Round-robin policy is unable to balance the system due to lack of consideration for numerous other factors as discussed earlier. As the trace file from IOR benchmark generates a continuous stream of file write requests instead of a more complex bursty pattern, the system still becoming unbalanced shows the weakness of the default approach to capture the application behavior and create a well-balanced and application-attuned load distribution.

This study highlights the need for better load balancing for the HPC I/O subsystem. Moreover, an opaque round-robin approach is incapable of accounting for workload dynamicity [27] such as that created by regular purge of old data in HPC systems (by OLCF practice [2]). In this paper, we address such problems by designing an OST management

Component	Factors	Discussion

Metadata Server (MDS)	`CPU%`
	`Memory%`
	`/proc/sys/lnet/stats`
		CPU and memory utilization reflect the system load.
		Load on the Lustre networking layer connected to MDS.

Metadata Target (MDT)	`mdt.*.md_stats` `mdt.*.job_stats`	Overall metadata stats per MDT. Metadata stats per MDT per job.

Object Storage Server (OSS)	`CPU%`
	`Memory%`
	`/proc/sys/lnet/stats`
		Reflects the system load of the management server.
		Load on the Lustre networking layer connected to OSS.

Object Storage Target (OST)	`obdfilter.*.stats`
	`obdfilter.*.job_stats`
	`obdfilter.*OST*.kbytesfree`
	`obdfilter.*OST*.brw_stats`
		Overall statistics per OST.
		Statistics per job per OST.
		Available disk space per OST.
		I/O read/write time and sizes per OST.

TABLE I: List of I/O performance statistics for relevant system components.

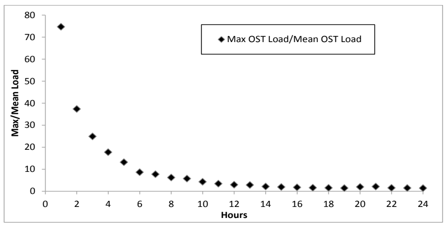

Fig. 1. Max/Mean OST load over time (Round-Robin).

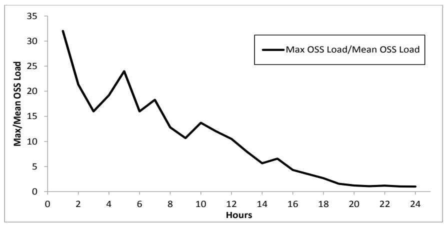

Fig. 2. Max/Mean OSS load over time (Round-Robin).

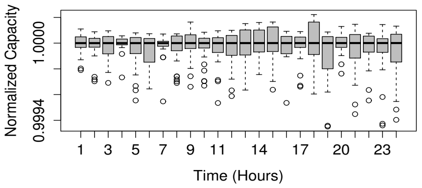

Fig. 3. Capacity of all OSTs over time (Round-Robin).

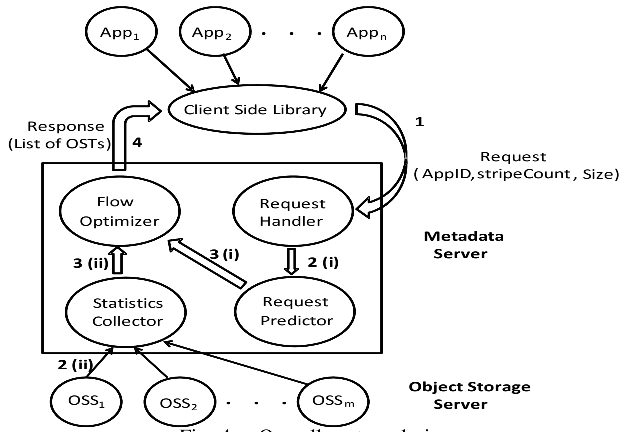

Fig. 4. Overall system design.

layer that provides a global and detailed system state view to the allocator.

III. System Design

In this section, we present the design of our load balancing system. We focus on Lustre due to its popularity in the top supercomputing systems, but our approach is also applicable to other distributed storage systems that use a hierarchical design.

Figure 4 shows the overall design of our proposed approach. Applications use a client side library that interfaces with a request handler on the MDS to handle application requests. Each request consists of an application ID, number of stripes, and the size of request.

Once a request arrives at the handler, we use a prediction model to predict future requests. It has been observed that different HPC applications have varying but predictable characteristic request pattern behavior [28]. We exploit this observation coupled with Markov Chain Model [23] to predict expected future requests from the applications and allocate resources accordingly. We capture system state via statistics collected using a scalable, distributed and hierarchical publisher-subscriber architecture. The OSSes act as publishers to provide statistics of OSSes and OSTs to MDS that acts as the subscriber. Then we design an adaptive load balancing algorithm that uses information about the state of various I/O servers and interconnects, as well as a set of predicted requests. The goal of our dynamic minimum-cost maximum-flow algorithm [4] is to determine a list of suitable OSTs to support the current and predicted requests in a load-balanced manner. Finally, the list of OSTs is provided to the application. The request handler, request predictor, flow optimizer and statistics collector are co-hosted and executed on the MDS. These components are not part of the Lustre MDS stack.

A. Framework for Statistics Collection

A key challenge in designing a load-balanced OST setup is realizing a scalable load monitoring solution. To this end, we adopt a hierarchical design that delegates the responsibility of load monitoring across multiple layers. Figure 5 shows a typical Lustre deployment, instrumented with our monitoring software probes that collect the statistics about the various I/O components of a Lustre deployment.

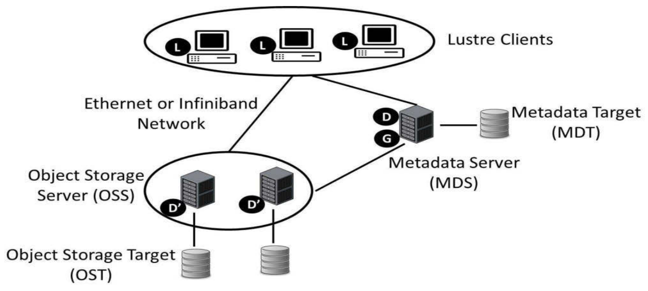
Lustre Clients
L
L
L
Ethernet or Infiniband Network
D
G
Metadata Target (MDT)
Metadata Server (MDS)
Object Storage Server (OSS)
D'
D'
Object Storage Target (OST)

Fig. 5. Monitoring system architecture.

We utilize a publisher-subscriber architecture [19] for statistics collection with the OSSes acting as publishers and the MDS acting as the subscriber. Publisher (OSS) and subscriber (MDS) are interconnected with each other using Ethernet or Infiniband network. Storage targets are connected to the servers using traditional storage network or direct attached storage technologies. Table I shows the list of statistics that we use in our approach. The D' component (shown in Figure 5) on an OSS acts as the statistics collector for both the OSS as well as the associated OSTs. Once the OSS and OSTs' data is collected, it is parsed and published to the publisher-subscriber system via a message queue, from where the MDS subscriber can retrieve the data.

D is the statistics collector component on MDS and monitors both the MDS and the associated MDTs. Similar to D' on OSS, D collects LNET statistics, CPU and memory utilization of MDS, and statistics for MDTs.

To ensure proper operation at scale and to avoid being overwhelmed by a large number of statistics from many Lustre components, we employ a message broker that logically sits between the OSSes and MDS, and collects data from the publishers. The broker then uses a common queue to communicate with the MDS. This allows the MDS to retrieve messages from the common queue at its own pace, and avoid lost messages. Moreover, big data applications typically issue about $10,000$ file events per second with most events being read-intensive [32]. Thus, to avoid network overloading, we set the statistic collection interval to be longer (5 seconds in our setup). To provide scalability and reduce monitoring overhead, we piggyback our messages on the already existing system monitoring infrastructure, such as ODDMON [18], that is typically already deployed in large supercomputing facilities for system health monitoring and management. The approach offers a promising low-cpu ($< 0.8\%$) and memory ($< 12\ MB$) overhead on MDS and OSSes, as we show using a real cluster setup in Section IV-A2.

B. *Applying Machine Learning for OST Request Prediction*

HPC applications have been shown to exhibit distinct patterns [21], [28]. Given the longevity of legacy applications, and our interactions with HPC practitioners, similar behavior predictability is expected for emerging applications as well. Therefore, application behavior can be modeled to help predict future application requests for OST allocations. We leverage this by identifying two key predictable properties of HPC applications' I/O requests namely, the number of requested OSTs (stripe count) and the number of bytes to be written in each request.

We balance the load during file creation and use the *number of bytes to be written* in the request. If the request is a read request, the number of bytes written will be zero. We focus on writes more than reads because once the file writes have been distributed, the caching mechanism and approaches such as burst buffers can absorb much of the read requests. Consequently, read performance is mostly shielded from the load imbalance at the OSTs, and writes are the overwhelming factor to be considered [27].

We use a Markov Chain Model [23] to capture the two properties (stripe count and bytes written) of application requests, factor in the current patterns, and predict future requests. The predicted requests along with the current application requests are then used to drive our load balancer.

Markov chain is a process in which the outcome of an experiment is affected by the outcome of a previous experiment. If a future state in the model depends on the previous *m* states, it is termed as a Markov chain of order *m*. More memory can be built into the model by using a higher order Markov model. But, as the order of Markov chain model becomes higher, the estimates of different parameters become less reliable. Also, the model becomes complex for higher orders with exponentially increasing computation time.

Figures 6, 7 and 8 show the predictions from Markov chains of varying orders on a collective trace of three different applications running simultaneously on the system. Here again, the application traces are from IOR benchmark [13], HACC I/O Kernel [12], and Transaction Processing Application (TPA) [26]. The results for orders 1 and 2 show that predicted write bytes are different from the actual bytes requested to be written by the application. We get the best results for order 3. We tested our model for higher orders as well, and the results are shown in Table II. All results are from the average of three runs. The results show that while the increase in accuracy from using a higher order model chain (greater than 3) was minimal, the corresponding execution time became very high. Thus, we do not go beyond third order chains in our approach.

We assume that the stripe size for all requests in a particular application is the same [27]. Thus, we can get the stripe count by predicting just the number of bytes to be written by a request. Dividing the number of write bytes by the stripe size gives us the stripe count. Here, we model the number of bytes to be written, with associated states shown in Figure 9. Over time, the number of bytes written can be modeled as a normal distribution with mean μ and standard deviation σ. The normal curve can be divided into five distinct segments (mean μ, between μ and ± 1 standard deviation, and between ± 1 standard deviation and ± 2 standard deviation). This is because 68.3% of the data under the normal curve falls between ± 1 standard deviation, and 95.4% of the data falls between ± 2 standard deviation. But to uniformly distribute the normal

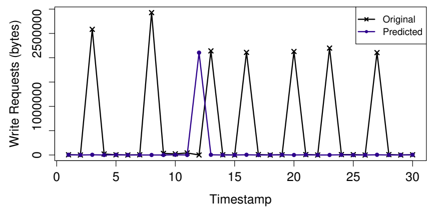

Fig. 6. Orig. vs. predicted requests (Order = 1).

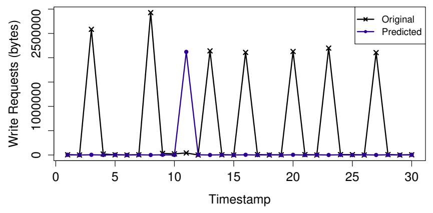

Fig. 7. Orig. vs. predicted requests (Order = 2).

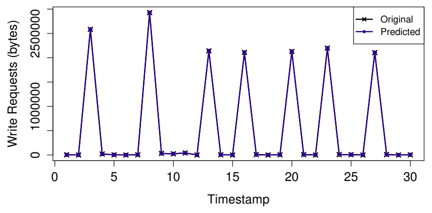

Fig. 8. Orig. vs. predicted requests (Order = 3).

Order of Markov chain	1	2	3	4	5	6	7	8
Execution time (seconds)	0.01	0.02	0.15	1.0	2.6	3.8	5.6	8.0
Accuracy (%)	65.6	78.7	95.5	96.3	96.8	97.2	97.5	97.8

TABLE II: Execution time and accuracy % for Markov chain for varying orders.

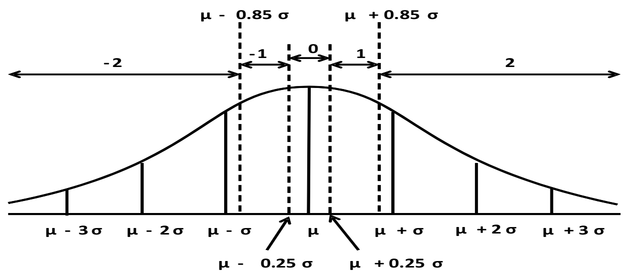

Fig. 9. States for write bytes.

curve, we divide the area into 5 segments such that each has an equal area under the curve (0.2). We assign states to the bytes to be written (R_i: $i^{th}\, request\, of\, the\, input\, set\, of\, requests$) using the function $assignStates$ in Algorithm 1.

After states have been assigned for the whole input request set, the training process (function $trainModel$ in Algorithm 1) starts in one of the states S'_i and moves onto the next state S'_j with probability p_{ij}. These probabilities are the transition probabilities and are captured by training the model based on the requests observed so far. We utilize dynamic learning, wherein the probabilities are continuously updated as more requests are processed. For example, when predicting the number of bytes to be written, every request that arrives is stored as a key with value as the requests that follow. Since, our model is a Markov chain of order 3, we store three following requests. The training sequence of requests is a list of any length longer than the order of Markov chain, from which the model draws state information for the chain. Our model dynamically determines the length of the training sequence. The dynamic approach is needed to accommodate the fact that the request size and pattern will vary for different applications as well as over time. We capture such changes using mean square error computation. To this end, we find the mean square error for different lengths of training sequence (e.g., 2, 4, etc.) to arrive at a length that yields the least mean square error. The model then uses this computed length for training. We limit the length to 16 to keep the computational overhead of our approach in check. Our Markov model can learn different applications behavior and assign them different identifiers. The model differentiates between the applications based on their assigned identifiers and predict the appropriate set of application-specific requests.

After the model is trained, we predict the states of future requests (function $predictModel$). Next, we have to map the states of the requests to the predicted write bytes. The function $statesToPrediction$ shows how the five different states are mapped. The values 0, 0.5, and 1.7 in the mapping are the quantized values in the five regions of the normal distribution curve, that is, the values where a normal line meets the mean slope of the respective region in the curve. Since the stripe size is fixed, once we determine the number of write bytes for the predicted set of requests, we can calculate the stripe count as well.

We employ a new client-side library (L in Figure 5) that the Lustre clients can use to request OSTs during file creation. L interacts with the global mapper G on MDS to obtain the set of OSTs on which the file is to be created. G monitors the load on MDS, and whenever the load is below a pre-specified threshold (CPU usage $<$ 70% in our implementation), the system runs the prediction algorithm, and uses the trained Markov model to create a prediction set with expected future requests. The current and predicted request sets are then forwarded to the load balancer, thus completing the prediction process for the current round of predictions. We test the model using a real cluster setup in Section IV-A2 and show that cpu ($< 4.5\%$) and memory overhead ($< 90\ MB$) remains small even with multiple applications. Thus our model scales well with increasing number of applications.

C. Load Balancing Algorithm

The completion of the monitoring and prediction phases provide us with the information to drive our load balanced allocation scheme for OSTs. We model the allocation problem as a minimum-cost maximum-flow (MCMF) optimization [9] over a flow network. The problem aims to find the maximum flow with the minimum cost from source to sink. We first explain the flow network structure and then discuss how the MCMF algorithm works.

In contrast to other approaches, e.g., maximum flow algorithms, MCMF enables us to capture the cost of flow for each edge, as well as enforce the needed strict bounds on the flow for both the source and the sink. As sources and sinks represent actual clients and OSTs, respectively, the limit enforcement is needed for load balancing by preventing load on an OST from varying too much from that of other OSTs. The selection of

Algorithm 1: Markov Chain Prediction.

```
Input: Set of current requests' write bytes $S$
Output: Set of predicted requests' write bytes $P$
begin
    $\mu$ = Calculate mean for $S$
    $\sigma$ = Calculate standard deviation for $S$
    $S'$ = assignStates (S, $\mu$, $\sigma$)
    $T$ = trainModel ($S'$)
    $P'$ = predictModel ($T$)
    $P$ = statesToPrediction ($P'$, $\mu$, $\sigma$)
    return P
Function assignStates
    Input: input set $S$, mean $\mu$, standard deviation $\sigma$
    Output: set of states $S'$
    for Request $R_i$ in $S$ do
        if $\mu - 0.25\sigma \leq R_i \leq \mu + 0.25\sigma$ then
            $S'_i$ = 0
        else if $\mu - 0.85\sigma \leq R_i < \mu - 0.25\sigma$ then
            $S'_i$ = -1
        else if $\mu + 0.25\sigma < R_i \leq \mu + 0.85\sigma$ then
            $S'_i$ = 1
        else if $R_i < \mu - 0.85\sigma$ then
            $S'_i$ = -2
        else
            $S'_i$ = 2
    return ($S'$)
Function statesToPrediction
    Input: predicted set of states $P'$, mean $\mu$, standard
           deviation $\sigma$
    Output: set of predicted requests' write bytes $P$
    for State $St_i$ in $P'$ do
        if $St_i = 0$ then
            $P_i = \mu$
        else if $St_i = -1$ then
            $P_i = \mu - 0.5\sigma$
        else if $St_i = 1$ then
            $P_i = \mu + 0.5\sigma$
        else if $St_i = -2$ then
            $P_i = \mu - 1.7\sigma$
        else
            $P_i = \mu + 1.7\sigma$
    return ($P$)
```

a lowest-cost path between source and sink by MCMF also ensures that the intermediate components, e.g., OSSes and LNET Routers, are not overloaded in quest of load-balanced OSTs. Thus, applying MCMF enables an optimal load balance across OSSes, OSTs and LNET Routers.

1) Flow Network: A flow network is a directed graph where edges from the source to the sink nodes carry the flow. Every edge in the flow network has an associated cost and capacity. Figure 10 shows the formation of the flow network when adapted to our target case.

In the network, S_1 and S_2 are the source and sink nodes, respectively. $R_1, R_2, ..., R_n$ are the combination of current and predicted application requests. The cost of the edges between source node and application requests is zero, and the capacity of the edges are the number of stripes for a particular request.

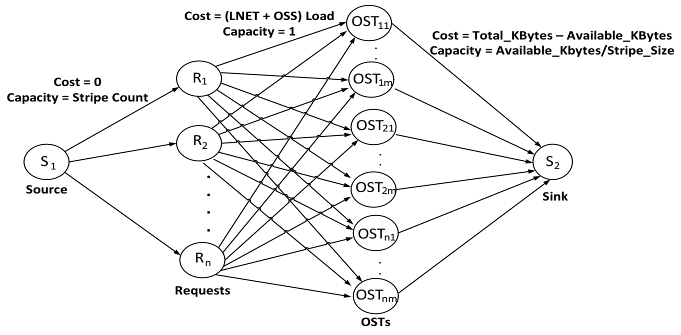

Fig. 10. Flow network used in the MCMF algorithm.

Every request is connected to all the OSTs, as any OST can potentially be used to service the request. The cost of the edge between a request (R_i) and an OST_{jk} is shown as the cumulative load on OSS_j on which OST_{jk} resides and the Lustre networking statistic involved between the MDS and OSS_j. The load on OSS_j can be expressed as $\alpha(CPU\%_j) + \beta(Memory\%_j)$, where α and β are the weights associated with CPU and memory utilization, respectively. We assign both values as 0.5, thus giving equal weights to both[1]. The capacity of the edge between a request (R_i) and an OST_{jk} is 1 so that it follows the feasibility constraint of our problem. The feasibility constraint specifies that a single OST can only service one stripe of a particular request. This ensures that the stripes are indeed distributed across OSTs.

Finally, all OSTs are connected to the sink (S_2), which is responsible for maintaining a load balanced setup. The cost of the edge from OST_{jk} (OST_k of OSS_j) is the load on OST_{jk}, shown as $TotalKB_{jk} - KBavailable_{jk}$. The capacity of the edge between vertices OST_{jk} and sink S_2 is the maximum number of stripes that the OST can hold. We assume that the stripe size for all the stripes for every request of an application is the same [27]. This is a practical assumption as stripe size is typically a system-wide configuration parameter and a characteristics of the storage I/O stack. Therefore the capacity of the edge will be the ratio of the available space on the OST to the constant stripe size.

2) Minimum-cost Maximum-flow (MCMF) Algorithm: The input to MCMF is a directed flow network $G = (N, A)$ as explained in Section III-C1. Every edge $(i, j) \in A$ has an associated cost $cost_{ij}$ and capacity cap_{ij}. MCMF aims to find the optimal flow from all requests (current and predicted) to the sink satisfying the capacity constraint on each edge.

More formally, the specific goal of our application of the MCMF algorithm [11] is to find $flow$ such that Equation 1 is minimized, while satisfying the capacity constraint expressed by Equation 2.

$$\sum_{(i,j)\in A} \cos t_{ij}\, flow_{ij} \qquad (1)$$

$$0 \leq flow_{ij} \leq cap_{ij}, \forall (i, j) \in A \qquad (2)$$

[1]The weights can be modified to accommodate applications needs, e.g., if one resource has higher constraints than other.

Finally, we use the Ford-Fulkerson Algorithm (FFA) [9] to solve the min-cost max-flow problem. Our choice of FFA is dictated by the elegance of its implementation, which enables it to scale. FFA runtime is bounded by $O(Ef)$, where E is the number of edges and f is the maximum flow of the graph. It has been shown that the FFA solver performs at least $n/14$ times faster than other MCMF solvers, where n is the number of nodes in a dense network [31].

FFA uses the concept of residual network that indicates additional possible flow in the network. Every edge $(i, j) \in A$ in the residual network is replaced by two edges (i, j) and (j, i). Edge (i, j) has cost $cost_{ij}$ and residual capacity $res_{ij} = cap_{ij} - flow_{ij}$. Additionally, edge (j, i) has cost $-cost_{ij}$ and residual capacity $res_{ji} = flow_{ij}$. Augmenting paths are found in the residual network, which adds to the flow in the graph. The augmenting path that is responsible for yielding the minimum cost is selected first and the request is mapped onto the OST on that path. The next minimum cost path is then selected and this continues until all the requests have been assigned to the OSTs. This completes the current allocation round, while ensuring even load distribution across the OSTs. The edges between OST_{jk} and sink S_2 are responsible for creating a load balanced distribution of OSTs. Additionally, if a flow is not possible for all the requests, we remove one of the predicted requests from the flow network given as input to the MCMF algorithm and run the algorithm again. This process is repeated until all requests are mapped onto the OSTs. The MCMF algorithm is implemented using the `networkx` package in python, and provides a low-overhead ($CPU < 2.3\%, Memory < 44\ MB$) solution (Section IV-A2).

D. Discussion

An important aspect of our design is that the OST selection process is repeated numerous times on the MDS to achieve the desired load balancing. Consequently, it is crucial that the load on the MDS does not become overwhelming. We take a number of steps to limit and reduce the impact on MDS performance. First, instead of passing the raw OSS statistics to the MDS, which would consume high network, message queue, and MDS resources, we parse OSS/OST information at the associated OSS, and reduce and compress the information before sending it to MDS. This distributes the load across OSSes, keeps the statistics processing on MDS in check, as well as reduces the pressure on the message queue. Second, we run the Markov chain model and the MCMF algorithm only when the CPU utilization on MDS (due to the normal operations of the MDS) goes below a specified threshold (70% in our implementation). This allows the MDS server to perform standard Lustre operations as a first priority. Third, the Markov chain model processing is independent of the number of applications concurrently running on the system as explained in Section III-B. So the overhead is not expected to increase as the number of applications increases. In terms of decision quality, MDS receives `brw_stats` of all the OSTs via the publisher-subscriber model. Therefore, it also knows the I/O read/write time per OST. This helps in removing the stragglers (i.e., slow OSTs) from the decision making list for allocation of requests. Moreover, based on interactions with HPC practitioners, we found that Lustre has a very steep learning curve and is often installed on systems where legacy and stability requirements entail very long upgrade cycles. Thus, an additional benefit of our approach without modifying core Lustre is that it gives users the flexibility to utilize our services quickly and with a greater degree of flexibility.

IV. Evaluation

We evaluate the efficacy of our approach using both a Lustre simulator and a live setup. In the following, we first describe our simulator and experimentation methodology, then compare our MCMF-based load balancing with the default Lustre OST allocation approach.

A. Methodology

1) Simulator: We have developed a discrete-time simulator based on the overall system design shown in Figure 4 to test our approach at scale. The simulator has four key components closely mirroring those of Lustre's OST, OSS, MDT, and MDS, which implement the various Lustre operations and enable us to collect data about the system behavior. The MDS is also equipped with multiple strategies for OST selection, such as round-robin, random, and MCMF. We have implemented a wrapper component that enables communication between our various simulator components. The wrapper is responsible for processing the input, managing the MDS, OST, and OSS communication and data exchange, and driving the simulation. All the network components in the simulator are modeled using Network Simulator (NS-3) [17]. The application traces collected from client side are modeled as clients in the simulator. In our simulations, all initial conditions are the same at the start of any allocation strategy. The parameters, number of OSSes, number of OSTs under each OSS are provided as inputs to the simulator.

2) Cluster Setup: We also conducted experiments on a small Lustre 2.8.0 setup to determine how the various components interact. The client node has 8 cores, 2.5 GHz Intel processor, 64 GB memory, and 500 GB HDD. MDS and two OSSes have 32 cores, 2.0 GHz AMD processors, 64 GB memory, and 1 TB HDD. All components are connected through a 10 Gbps Ethernet interconnect. We have set 6 OSTs in each OSS, each with 170 GB available disk space and 1 MDT in the MDS with a disk space of 100 GB. For real setup test, we repeated each experiment three times, and report the average results.

3) Workloads: To drive our simulations, we collect application traces at the client side. These application traces contain two kinds of data: (a) write entries, which have the timestamp, the number of bytes to be written, and the number of OSTs to be selected (i.e. the stripe count); and (b) read entry, which has the timestamp, number of bytes to be read and the OST ID from which the bytes have to be read.

To model the behavior of a real Lustre deployment, we run and capture a trace of 3 simultaneously running big data hpc applications on a production Lustre deployment. We use the HACC I/O kernel [12] that measures the I/O performance of the system for the simulation of Hardware Accelerated Cosmology Code (HACC) generating around 12000 file events per second, and the IOR benchmark [13] that is used for testing the performance of parallel file systems which generates around 20000 events per second. The third trace is generated from a high performance computing transaction processing application running at a large financial institution [26]. This trace generates about 15000 file events per second. For our tests (except the scalability study), we simulate the behavior of one MDS, eight OSSes with four OSTs per OSS, for a total of 32 OSTs. We also use our real-setup test to verify the results from the simulator.

B. Comparison of Load Balancing Approaches

We compare our MCMF based OST load balancing with the standard Lustre round-robin approach, as well as weighted random allocation where OSTs are selected at random from a subset of OSTs. Our random allocation model picks a random OST from a subset of the OSTs whose ratio of available space to total disk space is greater than 0.4. The goal is to remove the OSTs with less available space from the eligible list of OSTs to serve the request.

We measure the load balancing in our setup by plotting the ratio of the maximum disk space used to the mean disk space used over time. The ratio should tend to 1 for a load balanced setup. As seen from Figure 11, over a period of 24 hours, the random allocation gives the worst result as the ratio starts from 125, which is much more than the starting ratio in round-robin allocation. In contrast, we get better result with the MCMF allocation scheme. The ratio starts from 52 and has a steady decline to 1 in only 7 hours compared to 12 hours in round robin allocation.

In addition to balancing the load across OSTs, our objective is to also load balance the OSSes' CPU utilization—which is ignored under extant round-robin. Figure 12 shows the max/mean CPU utilization ratio of OSSes over a period of 24 hours. We see a better load balance in MCMF compared to round-robin and random allocations with a smoother decline of the ratio to 1 in 7 hours compared to 20 hours under round-robin strategy.

Our objective is to load balance OSTs such that every OST is at an almost similar state (in terms of number of bytes available, and load) under various file allocations. Since capacity of OSTs continuously decreases over time, we use normalized capacity where for every hour, the capacities of all OSTs are divided by the median capacity. Figure 14 shows the box-plot for normalized capacity vs. time for 23 hours under our approach. When comparing this with the round robin approach (Figure 3), we see that the inter-quartile ranges for MCMF are less wider than those for round-robin allocation. This shows that at any given time, MCMF is better able to balance load across OSTs.

We repeat the test on the real setup, and see a similar trend of Max/Mean load on both OSSes (CPU utilization) and OSTs (disk usage) as shown in Figures 11 and 12. Figure 15 shows the normalized capacity over time. The difference in the interquartile ranges in the box plots and a fewer number of outliers is due to the fact that on the real setup, we run 2 applications; IOR and HACC I/O kernel, compared to 3 applications being run on the simulator. The experiment on the real setup also shows that our approach gives better results on the real setup as well.

On the real setup, we also track on the MDS the CPU and memory utilization of our system components, i.e., the publisher-subscriber model, Markov model, and the minimum-cost maximum-flow algorithm. Figure 13 shows the maximum CPU utilization on MDS for a period of 12 hours over intervals of 30 minutes for our three components. The maximum CPU utilization is 1.3%, 3.8% and 6.5%, and the maximum memory utilization (not shown in a graph) on MDS are 15.2 MB, 75 MB and 200 MB for publisher-subscriber model, MCMF algorithm, and Markov model, respectively. On OSS, the CPU and memory utilization for the publisher never exceeds 1% and 12 MB, respectively. This shows that our approach requires negligible resources and can easily coexist with Lustre at scale. The Markov model component has the highest CPU and memory utilization, and to keep that in check, we designed the system to train the model and run MCMF algorithm only when the Lustre CPU utilization on MDS goes below a preset threshold (70% in our tests) to avoid any impact on data path performance.

C. Scalability Study

In our next experiment, we test how our approach will work with higher number of storage targets. For this purpose, we use a setup with an increasing number of OSTs from 160 to 3600.

In our simulations, in order to calculate the I/O bandwidth, all OSTs are assigned the same bandwidth at the start. The simulator also takes into account the number of applications using a particular OST at a given timestamp and calculates the read and write bandwidth accordingly. We assume that OSTs have equal read and write bandwidth. Figure 17 shows the overall mean I/O bandwidth for Lustre's default round-robin approach as well as our MCMF algorithm. As the number of OSTs increases, the mean bandwidth also increases for both the algorithms. Our algorithm provides better performance than round-robin solution even for higher number of OSTs. As seen from the figure, MCMF allocation is able to achieve up to 54.5% (under 800 OSTs) performance improvement compared to the default round-robin approach.

The overall execution time for load balancing using both round-robin approach as well as our algorithm (Markov model along with MCMF) is shown in Figure 16. Round-robin takes less time than our approach to allocate OSTs to incoming requests, but the difference between both execution times keeps on decreasing as we increase the number of OSTs. This is because, the increase in execution time with increase in the

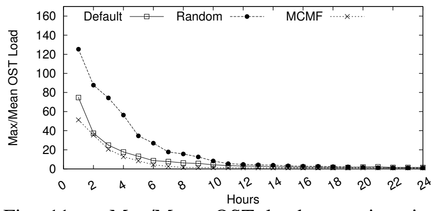

Fig. 11. Max/Mean OST load over time in

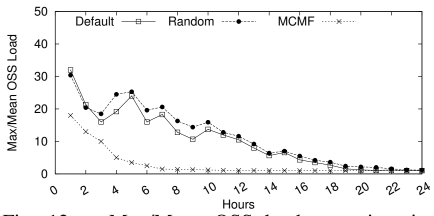

Fig. 12. Max/Mean OSS load over time in

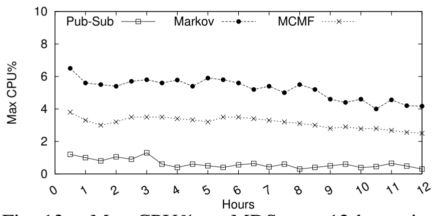

Fig. 13. Max CPU% on MDS over 12 hours in real setup.

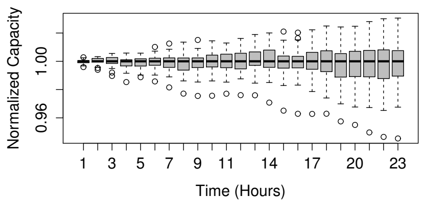

Fig. 14. Capacity of all OSTs over time under MCMF in simulated setup.

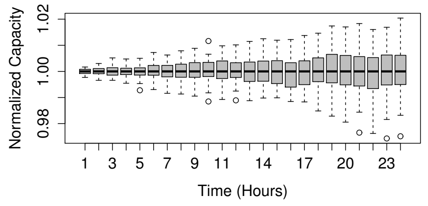

Fig. 15. Capacity of all OSTs over time under MCMF in real setup.

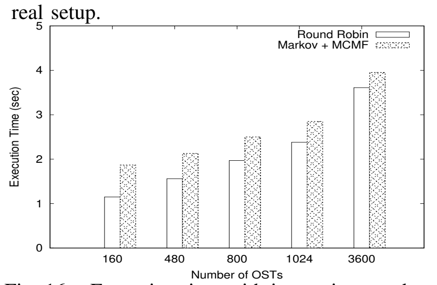

Fig. 16. Execution time with increasing number of OSTs in simulated setup.

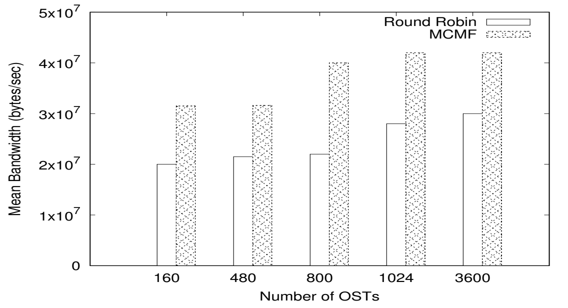

Fig. 17. Performance with increasing number of OSTs in simulated setup: 160 OSTs (20 OSS), 480 OSTs (60 OSS), 800 OSTs (100 OSS), 1024 OSTs (128 OSS) and 3600 OSTs (450 OSS).

number of OSTs is much higher in Round-robin than MCMF. This shows that our approach is more scalable. The gain in the overall performance as shown in Figure 17 is much higher than the gain in execution time, even for fewer number of OSTs.

The tests show that MCMF algorithm provides better load balanced allocation of OSTs with improved performance compared to Lustre's default round-robin allocation. Also, the performance of our algorithm does not degrade even with very large number of OSTs. Moreover, with higher number of OSTs, the execution time for our approach to allocate OSTs to requests is similar to that of Lustre's default round-robin approach. This is seen in Figure 17, where even for higher number of OSTs, MCMF performs better than the standard approach. Note that the difference in the CPU utilization on MDS when using the default round-robin allocation compared to when MCMF algorithm along with Markov chain model was being executed never exceeds 7.3%. Therefore, the benefits achieved by MCMF algorithm over standard round-robin allocation is achieved at a manageable cost, which is further amortized by the overall application I/O improvement resulting from the better load balanced setup.

V. Related Work

Load management has been incorporated into a number of modern distributed storage system designs. GlusterFS [3] uses elastic hashing algorithm that completely eliminates location metadata to reduce the risk of data loss, data corruption, and data unavailability. However, no load balancing is supported across the storage targets. Ceph [16], [29] uses dynamic load balancing based on CRUSH [30], a pseudo-random placement function. It also adds limited support for read shedding, where clients belonging to a read flash crowd are redirected to replicas of the primary copy of the data. However, the approach does not guarantee that the primary copies themselves are evenly distributed for optimum utilization of the storage resource. Our approach considers multiple factors and uses a global view of the system to make load balancing decisions for storage targets.

Several recent storage systems have explored optimization techniques for load balancing. In [6], dynamic data migration is proposed to balance the load under various constraints. Such approaches add the overhead of migration, while also maintaining availability and consistency. The VectorDot [25] algorithm is able to incorporate all these different constraints, as it is a multidimensional knapsack problem. It is suitable for hierarchical storage systems, as it can model hierarchical constraints. However, unlike our approach, the algorithm cannot effectively use history information for load balancing.

Machine learning and data mining techniques have also been used for the more general problem of resource allocation that also includes some load balancing. Martinez et. al. [14] introduce basic learning techniques for improving scheduling in hardware systems. These techniques are focused on individual hardware components and cannot be easily adapted to distributed file systems. A rule based approach to balance load in distributed file servers using graph mining methods is proposed in [10], where access patterns of files is used to relocate the file sets among different file servers. Schaerf et. al. [24] explore the problem space of adaptive load balancing using reinforcement learning techniques. Game Theory is also used for resource allocation [20]. These works are complementary to our approach, but require significant effort in feature selection and experimentation to enable such techniques in our target problem. We leverage such works in our approach to realize better I/O behavior capturing and improved predictions.

Finally, Google has recently explored machine learning to optimize various system-level metrics [15]. While such techniques show the promise of machine learning for optimizing system parameters, they are orthogonal to our target problem of load balancing in HPC storage systems, and not directly applicable to our use case.

VI. Conclusion

We have presented a load balancing approach for extreme-scale distributed storage systems, such as Lustre, where we enable the system to have a global view of the hierarchical structure and thus make more informed and load-balanced resource allocation decisions. We design a global mapper to be located in MDS of Lustre, which uses a publisher-subscriber model to collect runtime statistics of the various components in the I/O system by piggybacking the data on existing communication, employs Markov chain model to predict future application requests based on past behavior, and a minimum-cost maximum-flow algorithm to select OSTs in a load-balanced fashion. Experiments show that our approach provides a better load balanced solution for both OSSes and OSTs than the extant round-robin approach used in Lustre. This will lead to better end-to-end performance for HPC big data applications. In our future work, we plan to incorporate our solution into other systems such as Ceph and GlusterFS, as well as study our approach under different failure scenarios.

Acknowledgment

The authors would like to thank Ross Miller for his help in developing and evaluating the data collection mechanisms.

This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725.

This work is also sponsored in part by the NSF under the grants: CNS-1565314, CNS-1405697, and CNS-1615411.

References

[1] A. S. Bland, J. C. Wells, O. E. Messer, O. R. Hernandez, and J. H. Rogers. Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory. In *Proceedings of Cray User Group Conference (CUG 2012)*, May 2012.

[2] B. Bland. Titan-early experience with the titan system at oak ridge national laboratory. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 2189–2211. IEEE, 2012.

[3] E. B. Boyer, M. C. Broomfield, and T. A. Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Laboratory (LANL), 2012.

[4] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[5] P. J. Braam and R. Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[6] S. C. Deshmukh and S. S. Deshmukh. Improved load balancing for distributed file system using self acting and adaptive loading data migration process. In *4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions), 2015*, pages 1–6. IEEE, 2015.

[7] B. Dong, X. Li, Q. Wu, L. Xiao, and L. Ruan. A dynamic and adaptive load balancing strategy for parallel file system with large-scale i/o servers. *Journal of Parallel and Distributed Computing*, 72(10):1254–1268, 2012.

[8] J. Dongarra, H. Meuer, and E. Strohmaier. Top500 supercomputing sites. http://www.top500.org, 2016.

[9] L. R. Ford Jr and D. R. Fulkerson. A simple algorithm for finding maximal network flows and an application to the hitchcock problem. Technical report, DTIC Document, 1955.

[10] A. Glagoleva and A. Sathaye. Load balancing distributed file system servers: a rule-based approach. *Web-Enabled Systems Integration: Practices and Challenges: Practices and Challenges*, page 274, 2002.

[11] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: fast, centralized cluster scheduling at scale. In *Proceedings of OSDI16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 99, 2016.

[12] LLNL. Hacc i/o benchmark summary, 2017.

[13] LLNL. Ior benchmark, 2017. https://asc.llnl.gov/sequoia/ benchmarks/IOR_summary_v1.0.pdf.

[14] J. F. Martinez and E. Ipek. Dynamic multicore resource management: A machine learning approach. *IEEE Micro*, 29(5):8–17, 2009.

[15] R. Miller. Google using machine learning to boost data center efficiency — data center knowledge, 2014.

[16] E. Molina-Estolano, C. Maltzahn, and S. Brandt. Dynamic load balancing in ceph. 2008.

[17] NS-3. Network simulator, 2017. http://code.nsnam.org/.

[18] ORNL. Oddmon, 2017. https://github.com/ORNL-TechInt/oddmon.

[19] A. K. Paul, R. Chard, K. Chard, S. Tuecke, A. R. Butt, and I. Foster. Toward scalable monitoring on large-scale storage for software defined cyberinfrastructure. In *2nd Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS), 2017*. IEEE, 2017.

[20] A. K. Paul and B. Sahoo. Dynamic virtual machine placement in cloud computing. In *Resource Management and Efficiency in Cloud Computing Environments*, pages 136–167. IGI Global, 2017.

[21] A. K. Paul, W. Zhuang, L. Xu, M. Li, M. M. Rafique, and A. R. Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 110–119. IEEE, 2016.

[22] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger. A novel network request scheduler for a large scale storage system. *Computer Science - Research and Development*, 23(3):143–148, 2009.

[23] R. R. Sarukkai. Link prediction and path analysis using markov chains. *Computer Networks*, 33(1):377–386, 2000.

[24] A. Schaerf, Y. Shoham, and M. Tennenholtz. Adaptive load balancing: A study in multi-agent learning. *Journal of Artificial Intelligence Research*, 2:475–500, 1995.

[25] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of ACM/IEEE SC*, 2008.

[26] UMass. Umass trace repository, 2017. http://traces.cs.umass.edu/ index.php/Storage/Storage.

[27] F. Wang, S. Oral, S. Gupta, D. Tiwari, and S. S. Vazhkudai. Improving large-scale storage system performance via topology-aware and balanced data placement. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 656–663. IEEE, 2014.

[28] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. Long, and T. T. McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, 2004.

[29] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[30] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of ACM/IEEE SC*, 2006.

[31] Zealint. Maximum flow: Augmenting path algorithms comparison, 2017. https://www.topcoder.com/community/data-science/data-science-tutorials/maximum-flow-augmenting-path-algorithms-comparison/.

[32] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 61–70. IEEE, 2014.