Computing Load Aware and Long-View Load Balancing for Cluster Storage Systems

Guoxin Liu and Haiying Shen and Haoyu Wang
Department of Electrical and Computer Engineering
Clemson University, Clemson, SC 29631, USA
{guoxinl, shenh, haoyuw}@clemson.edu

***Abstract*—In large-scale computing clusters, when the server storing a task's input data does not have sufficient computing capacity, current job schedulers either schedule the task and transmit the input data to the closest server or let the task wait until the server has sufficient computing capacity, which generates network load or task delay. To handle this problem, load balancing methods are needed to reduce the number of overloaded servers due to computing workloads. However, current load balancing methods either do not consider the computing workload or assume that it is proportional to the number of data blocks in a server. Through trace analysis, we demonstrate the diversity of computing workloads of different tasks and the necessity of balancing the computing workloads among servers. Then, we propose a cost-efficient Computing load Aware and Long-View load balancing approach (*CALV*). In addition to the computing load awareness, *CALV* is also novel in that it achieves long-term load balance by migrating out data blocks from an overloaded server that contribute more computing workloads when the server is more overloaded and contribute less computing workloads when the server is more underloaded at different epochs during a time period. *CALV* also has a lazy data block transmission method to improve the load balanced state and avoid network load peak. Trace-driven experiments in simulation and a real computing cluster show that *CALV* outperforms other methods in terms of balancing the computing workloads and cost efficiency.**

***Keywords*-Computing cluster; Data allocation; Load balancing; Data locality**

I. Introduction

Large-scale computing-based storage systems, such as GFS [1] and HDFS [2], have been widely used to serve data-intensive computing frameworks (e.g., MapReduce [3]) in computing clusters to concurrently support a variety of data-intensive applications (e.g., search indexing, recommendation systems and scientific computation [4]). Data-intensive applications have a large amount of input data and a certain or even large amount of computing workloads. Sharing a cluster infrastructure among different applications facilitates data sharing among the applications and also enhances the resource utilization of the cluster, which saves the capital cost of building separate clusters for each kind of applications. However, the applications suffer unpredictable performance variations [3] due to the resource multiplexing between them.

Current computing clusters depend on job schedulers [2, 5, 4] of computing frameworks to improve system efficiency such as data locality and task delay. Preserving "data locality" is to put computing workloads, such as mapper tasks in MapReduce [3], with their input data. That is, when a task's data servers (i.e., the servers storing the task's input data) do not have sufficient computing capacities to host this task, it is allocated to the closest server with sufficient computing capacity [2, 5]. However, it cannot preserve data locality, which requires data transmission from the task's data server to its allocated server and generates network load. In order to preserve the data locality, the Delay scheduler [4] postpones a task's running until its data server has sufficient computing capacity, which however introduces extra delay for task execution. Therefore, job schedulers cannot preserve data locality exclusively without sacrificing the task delay. To improve system efficiency with data locality preservation and low task delay, the cooperation between the job scheduler and the load balancer is critical. In this load balancing, when a server is overloaded by its computing workloads, it moves some data blocks to another server to release the computing workloads of tasks targeting on these data blocks. Then, a task's data servers are unlikely to have insufficient computing capacities. However, previously proposed load balancing methods [2, 6–14] for cluster storage systems aim to balance I/O load but fail to consider the computing workload, which however is the bottleneck in data-intensive applications. The works in [9, 10] balance the number of blocks among servers with the assumption that the computing workload of a server is proportional to the number of blocks stored in the server. However, this assumption does not always hold true due to different popularity of stored data among servers [11] and heterogeneous computing tasks in different applications.

Computing load unawareness aside, previous load balancing methods aim to achieve the load balanced state at the time of load balancing after each time interval rather than for a long term. Then, a server may still be overloaded at some time epochs within a time interval since different data blocks introduce different computing workloads at each epoch in the time interval, and a block's computing workload varies over different epochs in the time interval. In this case, the periodical time interval for running load balancing must be small enough in order to maintain the load balanced state most of the time, which however leads to many load balancing operations and hence high overhead. Therefore, it is critical to balance the computing workloads targeting data blocks at each time epoch during a time interval, i.e., achieving long-term load balance.

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

Further, the load balancing operation itself must be cost-efficient and scalable. There are tens of thousands of servers and billions of data blocks in a commercial computing cluster today [15], and the scale increases rapidly over time. A load balancer needs to keep track of the workload for each data block [11, 13, 14] and schedules "data reallocation", in which data blocks are moved among servers to achieve load balance. A load balancer typically shuffles terabytes of data per day [10]. This information collection in tracking introduces tremendous network load and the reallocation scheduling generates high computing load to the load balancer. Then, the load balancer may become a bottleneck, which affects the efficiency of load balancing. Also, migrating many data blocks in a short time period in reallocation introduces a large peak network load to the storage system. To avoid this problem, a load balancer should reduce the number of data blocks being migrated at the same time. Therefore, designing a computing load aware load balancing method that meets the above requirements in a cluster storage system is not trivial.

In this paper, we first analyze a Facebook Hadoop cluster trace [15, 2], which shows that i) the computing workloads of tasks are heterogenous, ii) there are many server overloads due to insufficient computing capacities, and iii) the server overloads either exacerbate data locality or delay task execution. Therefore, it is important to consider computing workloads in load balancing. For this purpose, we propose a Computing load Aware and Long-View load balancing method (*CALV*) with high cost-efficiency and scalability in a cluster storage system.

CALV has a coefficient-based data reallocation method that conducts data reallocation among servers to balance the computing workload, i.e., avoid overloads and fully utilize the computing capacities of servers. An overloaded server is overloaded at some epochs while may be underloaded at other epochs in a time period. To achieve long-term load balance during the time period, we define a coefficient for data blocks in an overloaded server to represent their priorities to be selected to reallocate. Rather than reporting the load information of all blocks, each overloaded server selects its partial blocks to report to the load balancer to reallocate in order to become non-overloaded. Thus, the network load for information collection, the computing load on scheduling reallocation, and the network load for reallocation are reduced. *CALV* is also enhanced by a lazy data block transmission to avoid large peak network load and improve the load balanced state.

CALV is the first work that balances the computing workloads in the long term among servers by reallocating data blocks among them. We summarize our contributions below.

- *Trace analysis on computing workloads.* We measured the Facebook Hadoop cluster trace to show the importance of computing load aware load balancing.
- *Computing load Aware Long-View load balancing method (CALV).* It consists of the following two components.

(1) *Coefficient-based data reallocation;*
(2) *Lazy data block transmission.*

- *Trace-driven experiments.* Trace-driven experiments in simulation and a real cluster show the effectiveness of *CALV* in balancing the computing workloads and its high cost-efficiency.

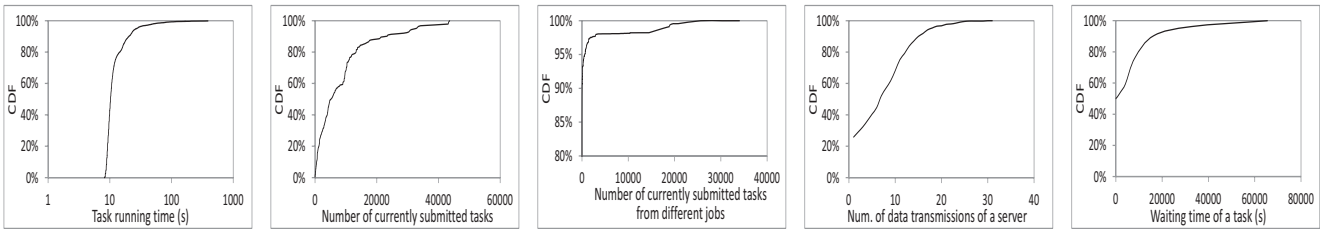
The rest of this paper is organized as follows. Section II presents the preliminaries of the load balancing problem and the trace analysis results. Section III presents the design of *CALV* in detail. Sections IV and V present the performance evaluation of *CALV* in simulation and a real cluster. Section VI presents the related work. Section VII concludes this paper with remarks on our future work.

II. Computing Load Aware Load Balancing Problem

A. Preliminaries

In this section, we present the environment of the cluster storage system and its load balancing problem. In a cluster storage system, we use S to denote the set of all servers, and s_i to denote the i^{th} server. We use $C^c_{s_i}$ to denote the computing capacity of s_i represented by the number of computing slots [2], such as cores of CPUs. There are a set of files, each of which is split into several data blocks (a unit of storage) [2, 11, 8]. We use d_j to denote the j^{th} data block, and all data blocks have the same size [2]. We then use D_{s_i} to denote the set of all data blocks stored in s_i. To enhance data availability, each block has several replicas [2] stored in different servers. Since we focus on data-intensive computing applications containing long-term batch jobs, such as MapReduce jobs [3], the computing resource instead of I/O resource is the bottleneck of a server. Thus, we focus on balancing the computing workload in this paper. To additionally consider the I/O resources, we can easily add the I/O capacity, such as storage capacity of a data server, as constraints in data block reallocation.

Each data-intensive job, such as a MapReduce job [3], is constituted of tasks. A task such as a mapper [3], denoted by t_i, processes a data block using a certain computing resource. A task (or the computing workload of a task) can be denoted by a 3-tuple as $t_i =< e^s_{t_i}, e^f_{t_i}, C^c_{t_i} >$. $e^s_{t_i}$ and $e^f_{t_i}$ denote the time epochs during which t_i is submitted and finished, respectively, and $C^c_{t_i}$ denotes the required amount of computing resource of t_i, such as the number of computing slots in MapReduce. For the tasks that run periodically in a computing cluster [16], we can predict their execution time based on the historical log. For example, the Hadoop cluster for Facebook, Yahoo! and Google periodically process terabytes of data for advertisement, spam detection and so on [17]. For a new submitted job without an execution historical log, we can get its process execution time by profiling run [16, 18]. We also assume that each task's



(a) Running time of tasks (b) Num. of concurrently submitted tasks (c) Num. of concurrently submitted tasks from diff. jobs (d) Num. of data transmissions of a server (e) Waiting time of a task

Figure 1: Trace data analysis results.

submission time is predictable in advance according to its historical running records [16] or is indicated in advance. If a task runs multiple times in a time period, we consider them as different tasks associated with different running times. Then, $t_i =< e^s_{t_i}, e^f_{t_i}, C^c_{t_i} >$ of each task can be predicted.

We use T to denote a time interval for task scheduling and load balancing, and use e_k to denote the k^{th} time epoch during T. When a task is scheduled to its data server, the server does not have the task's required computing resource, we call such a server an overloaded server. The goal of our load balancing method is to reduce the number of such overloaded servers by data reallocation while achieving high cost-efficiency with low network load, which is measured by the product of the size of transmitted data and the transmission path length [19, 20].

B. Trace Data Analysis

None of the previous load balancing algorithms [8–14] considers the computing workloads associated with each data block for load balancing. The works in [9, 10] assume that the computing workloads in servers are proportional to the number of their stored data blocks, and aim to balance the number of data blocks among servers. In order to verify that this assumption does not hold true and the importance of considering the computing workloads in load balancing, we analyze a Facebook Hadoop cluster trace [15, 2]. It is a 24-hour job running trace that contains 24442 jobs, the submission times of the mapper tasks of each job, and the input, output and shuffle data size of each job. The number of tasks of a job varies from 1 to 87307. Each mapper task uses one computing slot for certain time to process one data block.

In order to find the running time of each task, we conducted a profiling run of the jobs in the trace in a Hadoop cluster with 128MB block size. The Hadoop cluster has 8 nodes in Palmetto [21], each of which has 8 cores and 32GB memory. We profiled the running time of each mapper task, which handles the inputs of a job. Figure 1(a) shows the cumulative distribution function (CDF) of the running time of all mapper tasks. We can see that the running time of different tasks varies significantly. It indicates that for different tasks, even though they require the same amount of computing resources and the same I/O resource (i.e., use one computing slot and request one 128MB data block), their computing workloads vary largely since they occupy the computing resources for different time periods. Therefore, simply balancing the number of I/O requests or the number of tasks targeting data blocks stored in servers cannot balance their associated computing workloads.

Then, we analyzed the trace and drew Figure 1(b) and Figure 1(c). Figure 1(b) shows the CDF of the number of concurrently submitted tasks in the trace. It shows that there are many tasks submitted concurrently. This implies that the tasks compete each other for the computing capacities on their data servers if their requested data blocks are stored in the same servers. Therefore, it is important to balance the computing workloads among servers over time to avoid server overloads.

Figure 1(c) shows the CDF of the number of concurrently submitted tasks belonging to different jobs in the trace. It shows that the tasks from different jobs may compete each other. Since different jobs have different data processing procedures, the mapper tasks in a server may generate different computing workloads. Therefore, simply balancing the number of tasks per server by balancing the number of data blocks processed by these concurrently submitted tasks [9] cannot solve the problem.

When a task's data server does not have sufficient computing capacity, i.e., when it is overloaded due to computing workload, the job schedulers handle this case in two ways. In the *FIFO* scheduler [2], the task is allocated to the closest server that has sufficient computing capacity and the data is transmitted from the task's data server to this server, which however generates network load. In the *Delay* [4] scheduler, the task waits until its data server has sufficient computing capacity, which however generates task delay. We then measure the number of such data transmissions and the task delay to show the adverse effect of computing workload imbalance.

We simulated 3000 servers as in [15] with 8 computing slots in each server and 10PB of data [22], which are randomly distributed among all servers. We simulated the 24442 jobs [15], and used the submission time and input/output sizes of a job in the trace. The requested data is randomly chosen and the execution time is set to the same time as in our profiling run. Figure 1(d) shows the CDF of the number of data transmissions from a server when it is overloaded with the *FIFO* scheduler. We see that there are more than 50% of all servers transmitting more than 6 data blocks to

other servers. It indicates that the data locality is exacerbated due to the server overloads. Figure 1(e) shows the CDF of the waiting time of all tasks. We see that there are around 50% of all tasks waiting for more than 16s. It indicates that the tasks are delayed due to the imbalance of computing workloads among servers. Figures 1(a) - 1(e) show that the computing load aware load balancing is very important to improve the data locality and reduce the task latency.

III. Computing Load Aware and Long-View Load Balancing

A. System Overview of CALV

Motivated by the observations from our trace study, we propose a Computing load Aware Long-View load balancing method (*CALV*) with high cost-efficiency in a cluster storage system. In *CALV*, each server periodically keeps track of the computing workloads targeting its data blocks at each epoch (e.g., the number of computing slots in each second) and creates the historical log. For new submitted tasks, the job scheduler notifies their allocated servers of their workloads. Based on the historical log and the notification, each server checks if it will become overloaded in the next T. Each overloaded server selects data blocks to migrate out to release its extra computing workloads while fully utilizing its computing capacity over T and reports the workloads of these blocks to the load balancer. Each server reports to the load balancer its computing workloads and computing capacity. Then, the load balancer schedules and conducts data reallocation for the reported data blocks from the overloaded servers. In the previous load balancing methods, each server reports the information of each data block to the load balancer. The pre-selection of migration blocks in *CALV* reduces the amount of reported data, and hence reduces the network overhead and the computing overhead in the load balancer.

One novelty of *CALV* lies in its coefficient-based data reallocation that helps achieve long-term load balance during T rather than at a time spot. During time period T, a server may be overloaded in some epochs while underloaded in other epochs. In an overloaded server, the data blocks that contribute more computing workloads when it is more overloaded and contribute less computing workloads when it is more underloaded at different epochs during T have higher priorities to migrate out. Thus, long-term load balance over T can be achieved with a limited number of data migrations.

CALV also incorporates one enhancement to improve the load balance performance and peak network load, a lazy data block transmission. Since the source server and destination server of a migration block may be overloaded at some epoches while non-overloaded at other epoches, the lazy data block transmission method delays the block migration until the source server is about to be overloaded due to the computing workloads targeting on this data block. This way, we can try to avoid the situation that the destination becomes overloaded by hosting this data block.

B. Computing Workload Tracking and Reporting

For long-term load balance, *CALV* aims to balance the computing workload at each epoch e_k in the next T. In the previous T, each server predicts the computing workloads targeting its stored data at each epoch in the next T. The computing workloads are predicted based on the historical logs for periodically running tasks and are notified by the job scheduler for new submitted jobs. To create the historical log, each server needs to record the workload of each task t_i requesting each of its data blocks at time e_k ($e^s_{t_i} \leq e_k \leq e^f_{t_i}$). The whole set of the tasks requesting data block d_j is denoted by $T^{e_k}_{d_j}$. Then, we can get the total workloads towards data block d_j at epoch e_k as:

$$L^{d_j}_{e_k} = \sum_{t_i \in T^{e_k}_{d_j}} C^c_{t_i}. \quad (1)$$

Recall that the whole set of data blocks stored in server s_i is denoted by D_{s_i}. Then, we can get the workloads on s_i during epoch e_k as:

$$L^{s_i}_{e_k} = \sum_{d_j \in D_{s_i}} L^{d_j}_{e_k}. \quad (2)$$

Then, for each server s_i, at epoch $e_k \in T$, if $L^{s_i}_{e_k} > C^c_{s_i}$ (i.e., the aggregated computing workload targeting on data blocks stored in s_i exceeds its computing capacity at epoch e_k), we regard s_i as an overloaded server at epoch e_k; if $L^{s_i}_{e_k} < C^c_{s_i}$, we regard s_i as an underloaded server at epoch e_k. A server s_i is an overloaded server if it is an overloaded server in at least one epoch in the next T. For the load balancer to schedule data reallocation, in the previous T, each server s_i reports its workload to the load balancer at each epoch as $L^{s_i}_{e_1}, L^{s_i}_{e_2}, ..., L^{s_i}_{e_n}$ and its computing capacity. Also, each overloaded server needs to pre-select migration data blocks to release its extra computing workload and report the workload of each of these blocks at each epoch as $L^{d_j}_{e_1}, L^{d_j}_{e_2}, ..., L^{d_j}_{e_n}$. There are a large amount of data blocks in the system and each block is associated with varying workloads over time. If the load balancer considers all the data blocks to achieve load balance, it cannot be scalable. Each overloaded server pre-selecting data blocks to reallocate increases the scalability of the load balancer.

C. Coefficient-based Data Reallocation

In this section, we first introduce how an overloaded server pre-selects data blocks to reallocate and then present how the load balancer schedules the data reallocation. We start with explaining the rationale of the migration block selection policy.

Rationale of the migration data block selection policy. Each overloaded server s_i selects partial of its data blocks to migrate out to release its extra computing overload. We use an example shown in Figure 2 to illustrate the rationale of our migration data block selection policy. In the figure, the

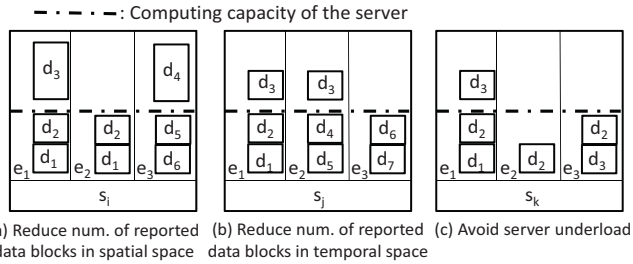


Figure 2: Selection of data block to reallocate.

height of each data block represents the computing workload targeting this data block during an epoch. For example, in Figure 2(a), the workload of data block 1 and 2 at epoch e_1 and e_2 equal to 1 computing slot and the workload of block 3 at epoch e_1 equal to 2 computing slots ($L_{e_1}^{d_3} = 2$). When an overloaded server selects the data blocks to migrate out, it follows two principles. We explain the principles below.

Each overloaded server should try to reduce the number of selected data blocks. This way, the size of information reported to the load balancer is reduced, so that the network load and computing load on the load balancer is reduced. Also, it reduces the reallocation overhead due to fewer block migrations. This objective can be achieved in both the spatial space and the temporal space. We define a server's *overloaded epoch*, *underloaded epoch* and *non-overloaded epoch* as the epoch in which the server is overloaded, underloaded and non-overloaded, respectively. The spatial space considers the workload of each block in one overloaded epoch. The temporal space considers the aggregated workload of each block in all overloaded epochs during T. In the spatial space, for example, in Figure 2(a), to release the extra 2 slots workload in e_1, d_3 (one block) should be selected rather than both d_1 and d_2 (two blocks). In the temporal space, for example, in Figure 2(b), d_3 should be selected to release the extra workload in both e_1 and e_2. Selecting any other block in e_1 and e_2 can only release the extra load of either e_1 or e_2. Therefore, the first principle is that *the data blocks contributing more computing workloads at more overloaded epochs in the spatial space and temporal space have a higher priority to be selected to reallocate.*

Each overloaded server also should try to fully utilize its computing resources, i.e., reduce the number of its underloaded epochs and its underloaded degree. By reallocating a data block to another server to release the extra load in an overloaded epoch, a server may become more underloaded at other epoches. For example, as shown in Figure 2(c), though reallocating d_2 or d_3 releases the extra load in epoch e_1, it makes the server more underloaded at epoch e_2 or epoch e_3, respectively. Therefore, d_1 should be reallocated instead of d_2 or d_3. Thus, the second principle of data block selection is that *among all data blocks contributing workloads at an overloaded epoch, the data blocks contribute less workload at more underloaded epochs have a higher priority to be selected to reallocate.*

In order to jointly consider the above two principles in migration data block selection, we introduce a load balancing coefficient for the blocks in an overloaded server to represent their priorities to be selected to reallocate. For an overloaded server s_i at epoch e_k, we define its unbalanced workload at epoch e_k as

$$u_{e_k}^{s_i} = \begin{cases} L_{e_k}^{s_i} - C_{s_i}^c & \text{if } L_{e_k}^{s_i} \neq C_{s_i}^c \\ -c & \text{otherwise} \end{cases} \quad (3)$$

where c is a computing capacity unit, such as a computing slot in MapReduce. We set it to $-c$ instead of 0 when s_i's computing resource is fully utilized in order to set a higher coefficient for d_1 than for d_2 and d_3 in Figure 2(c) according to the second principle. Then, we define the load balancing coefficient of data block d_j in server s_i as:

$$\sum_{\forall e_k \in T} u_{e_k}^{s_i} \cdot L_{e_k}^{d_j}. \quad (4)$$

The blocks with larger coefficients have a higher priority to reallocate. If $u_{e_k}^{s_i} > 0$ during e_k, which indicates that the server is overloaded, a larger $L_{e_k}^{d_j}$ leads to a larger coefficient. Therefore, it follows the first principle. Otherwise if $u_{e_k}^{s_i} < 0$, a smaller $L_{e_k}^{d_j}$ leads to a larger coefficient. Therefore, it follows the second principle.

For an overloaded server, releasing its extra computing workload is more important than fully utilizing its computing resource in load balancing. Therefore, the migration blocks should be selected from the blocks that contribute computing workloads at overloaded epochs within T. We then introduce a concept called overload contribution that measures the contribution of a block to the overload of an overloaded server. We first calculate the overload degree of an overloaded server s_i during each overloaded epoch e_k as

$$o_{e_k}^{s_i} = \begin{cases} L_{e_k}^{s_i} - C_{s_i}^c & \text{if } L_{e_k}^{s_i} > C_{s_i}^c \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

which shows how overloaded server s_i is during e_k. Then, we can calculate the overload contribution of a data block d_j stored in s_i by

$$\sum_{\forall e_k \in T} o_{e_k}^{s_i} * L_{e_k}^{d_j} \quad (6)$$

If the overload contribution is positive, it indicates that data block d_j introduces computing workloads at server s_i's overloaded epochs.

Next, we introduce the process for an overloaded server to select data blocks to report to the load balancer to reallocate. Server s_i first calculates the overload contribution of all data blocks stored in s_i. The server then chooses the blocks with positive overload contributions. For each of these blocks, server s_i calculates its coefficient based on Formula (4). The server then sorts these data blocks in a decreasing order of the coefficient. Starting from the first block in the sorted list, server s_i selects the blocks one by one in the top-down manner until it becomes non-overloaded at each epoch in T. Every time when a block, say d_j, is selected from the sorted list, the computing workload of s_i at each epoch e_k where

d_j contributes workload is updated by $L_{e_k}^{s_i} \leftarrow L_{e_k}^{s_i} - L_{e_k}^{d_j}$. Note that after reallocating the data blocks prior to data block d_j in the sorted list, the original overloaded epochs where d_j contributes computing workloads may become non-overloaded epochs. If all the original overloaded epochs where d_j contributes computing workloads become non-overloaded epochs, d_j is removed from the sorted list. This block selection process continues until server s_i is not overloaded during T. Then, the overloaded server s_i reports each selected data block d_j to the load balancer in the form of $L_{e_1}^{d_j}, L_{e_2}^{d_j}, ..., L_{e_n}^{d_j}$.

Data reallocation scheduling at the load balancer. At the previous T, each server checks if it will be overloaded in the next T based on predicted and notified computing workloads. If a server will be overloaded, it selects its migration data blocks and reports the information to the load balancer, as explained previously. Each server also reports its computing capacity ($C_{s_i}^c$) and computing workload at each epoch e_k to the load balancer ($L_{e_1}^{s_i}, L_{e_2}^{s_i}, ..., L_{e_n}^{s_i}$). The load balancer then schedules reallocating the reported data blocks to other servers that will not be overloaded or generate the least overload degree after hosting the blocks. Unlike the previous load balancing methods, we use all servers instead of underloaded servers as candidates to reallocate the reported data blocks. This is because overloaded servers in our method may have underloaded epochs in T before and during the reallocation scheduling, which can be allocated with workloads to reach the non-overloaded state in order to fully utilize the computing resource of the servers.

We hope to migrate the most loaded block to the least loaded server in order to quickly achieve load balance. Thus, the reported blocks are ordered in descending order of their accumulated workload during T, i.e., $\sum_{\forall e_k \in T} L_{e_k}^{d_j}$ and the underloaded servers are ordered in descending order of their accumulated available capacity during T, i.e., $\sum_{\forall e_k \in T}(C_{s_i}^c - L_{e_k}^{s_i})$. Starting from the first data block d_i in the sorted block list, the load balancer schedules to reallocate it from its source server s_i to another server to release the overloads caused by d_i at s_i. In the sorted server list, the load balancer checks each server in the top-down manner. For each picked server s_j, the load balancer first checks whether it has enough storage capacity of d_i and has no replica of d_i. If yes, the load balancer calculates the workload of s_j if it hosts d_i at each epoch e_k during T by $L_{e_k}^{s_j} \leftarrow L_{e_k}^{s_j} + L_{e_k}^{d_i}$. If s_j is non-overloaded at each epoch where d_i contributes computing workload (i.e., d_i's overload contribution to s_j equals to 0), d_i is scheduled to be reallocated to s_j. If such a server cannot be found in the sorted server list, the load balancer calculates d_i's overload contribution for each server according to Formula (6). It then chooses the server with the smallest overload contribution as block d_i's reallocated server. The smallest overload contribution to a server means that d_j contributes low workloads when that server is overloaded and d_j increases lower overload degree on this server than on other servers. Using this way, the load balancer schedules the reallocation of each block in the sorted block list to a server and finally completes scheduling the reallocation.

D. Lazy Data Block Transmission

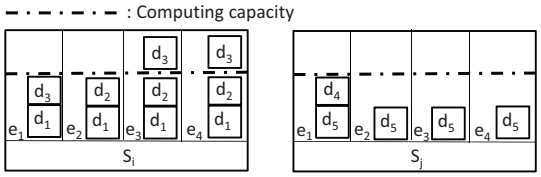


Figure 3: Lazy data block transmission.

The coefficient-based data reallocation method in Section III-C generates a new data allocation schedule offline, which means the actual data reallocation has not been conducted yet. Reallocating the data blocks right after the reallocation scheduling may introduce tremendous network loads in a short time and also overload the destination server, which may disturb the execution of user jobs at the beginning of next T. For example, in Figure 3, in the reallocation schedule, d_3 needs to be transmitted from s_i to s_j, since d_3 contributes more overloads in s_i than in s_j. If we transmit d_3 at the beginning of T at e_1, s_j becomes overloaded at e_1. However, if we wait and transmit d_3 at e_2, there will be no overloads for both s_i and s_j. Thus, to avoid overloading the destination server and the network load peak, the load balancer also determines the block transmission time. It delays the transmission of each block from the source server until the first time that the block contributes to the overload of the source server.

Below, we introduce how the load balancer determines the block transmission time for each data block to be reallocated. For data reallocation of each data block, say d_k, from serve s_i to server s_j, the load balancer finds their first overloaded epochs where block d_k contributes workload if they host d_k, denoted by e_{s_i,d_k}^o and e_{s_j,d_k}^o, respectively. That is,
$e_{s_i,d_k}^o = min\{e_r | e_r \in T \wedge L_{e_r}^{d_k} \cdot o_{e_r}^{s_i} > 0\}$;
$e_{s_j,d_k}^o = min\{e_r | e_r \in T \wedge L_{e_r}^{d_k} \cdot o_{e_r}^{s_j} > 0\}$.
If $e_{s_i,d_k}^o > e_{s_j,d_k}^o$, it means that s_i is not overloaded during data d_k's first several task processes, but s_j may be overloaded if reallocating d_k to it during this time period. The load balancer then calculates the completion time of the last task targeting d_k at s_j prior to e_{s_i,d_k}^o, denoted by $e_{s_j,d_k}^f = max\{e_r | e_r \in T \wedge e_r < e_{s_i,d_k}^o \wedge L_{e_r}^{d_k} \cdot o_{e_r}^{s_j} > 0\}$. Then, the load balancer randomly selects an epoch within $(e_{s_j,d_k}^f, e_{s_i,d_k}^o)$ for d_k's reallocation. Take d_3 in Figure 3 for example, we can get that $e_{s_i,d_3}^o = e_3$, $e_{s_j,d_3}^o = e_1$. Since $e_{s_i,d_3}^o > e_{s_j,d_3}^o$ and $e_{s_j,d_k}^f = e_1$, the data can be transmitted within (e_1, e_3), which is e_2. If $e_{s_i,d_k}^o \leq e_{s_j,d_k}^o$, it indicates that the source server is overloaded before the destination server becomes overloaded if it stores d_k. Then, d_k should be

transmitted before $e^o_{s_i,d_k}$. Thus, the load balancer randomly selects a time within $[e_1, e^o_{s_i,d_k})$ for d_i's reallocation.

IV. Trace-Driven Performance Evaluation

We conducted trace-driven experiments to evaluate *CALV* in comparison with other load balancing and data allocation methods using the Facebook Hadoop cluster trace [15, 2]. Based on the trace, we simulated 3000 servers in a computing cluster with the typical fat-tree topology [23]. In our experiments, we varied the number of jobs as x times of the number of jobs in the trace (i.e., 24442), where x was increased from 0.5 to 1.5 with a step size of 0.25. The submission time and the input/output size of each job were set to the values in the trace. As [16, 18], we set each task's execution time as its execution time in an offline running in Palmetto in Section II-B. The storage and computing capacities of each server were set to 12TB [24, 25] and 8 computing slots [21, 26], respectively. The default size of a block and the number of replicas were set to 64MB and 3, respectively [2]. The total size of all blocks was set to 10PB because there are tens of PBs of data in a commercial cluster such as the Facebook's cluster [22]. By default, each of the requested data block was randomly chosen from all blocks. We set the time period T to 24 hours and the epoch e to 1 second, and set $c = 1$.

We compared *CALV* with the following data allocation and load balancing methods: *Random* [2], *Sierra* [10], *Ursa* [11] and a Computing load Aware load balancing method (*CA*). *Random* randomly allocates data blocks to servers. *Sierra* allocates equal number of blocks among servers in order to balance the computing workload. *Ursa* allocates data blocks to servers so that each server has a request rate targeting its data blocks less than its I/O capacity. In our experiments, we modified *Ursa* to achieve an equal request rate on each server since we simulated a homogenous environment of servers with equal amount of each type of resources. *CA* uses the average computing workload per second to measure the block load in load balancing and aims to let all servers have the same average computing workload per second. We chose three typical job schedulers to assign tasks among servers after load balancing: *FIFO* [2], *Fair* [5] and *Delay* [4]. *FIFO* schedules jobs in an increasing order of their submission times and allocates a task to the closest server with sufficient computing capacity if its data server has insufficient computing capacity. *Delay* delays the scheduling of a task until one of its data servers has available computing slots. In *Fair*, tasks of different jobs share resource fairly, that is, currently running jobs have the same average number of computing slots over time. We use the FIFO scheduler by default.

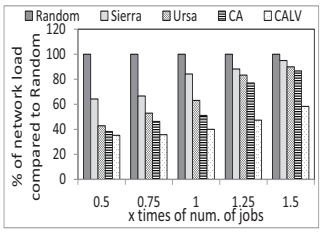
We first allocate data blocks to servers randomly. After running all the jobs with a job scheduler, we ran a load balancing method to reallocate data blocks. Then, we ran the jobs again and measure the performance. We reported the average performance of each method from 10 experiments.

A. Effectiveness of Load Balancing

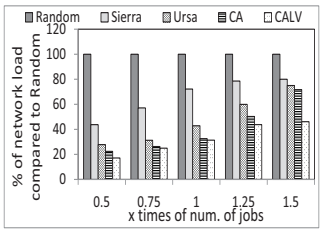
1) Performance of Data Locality: In the *FIFO* and *Fair* schedulers, when a task is allocated to another server when its data server does not have sufficient computing capacity, the task's required data is transmitted from its data server to its allocated server. We first show the network load due to such data transmissions, which is measured by the product of the size of transmitted data and the transmission path length [19, 20].

Figures 4(a) and 4(b) show the percentage of the network load of each load balancing method compared to the network load of *Random* using the *FIFO* and *Fair* schedulers, respectively. The result follows 100%=*Random*>*Sierra*>*Ursa*> *CA*> *CALV*. *Random* allocates data blocks to all servers randomly without a specific load balancing operation. Thus, more servers become overloaded due to computing workloads since they store more data blocks being processed by tasks simultaneously, leading to many data transmissions to other servers for processing. As a result, *Random* generates the largest network load. *Sierra* balances the workload by allocating equal number of data blocks to servers. However, data blocks may be processed by different number of tasks and different tasks require different amounts of computing resources. Therefore, some servers may still become overloaded and need to transmit data blocks to other servers. Thus, *Sierra* generates network load less than *Random* but larger than other methods. *Ursa* balances the average number of data requests per unit time among servers but does not balance the computing workloads among servers. Therefore, some servers may be overloaded due to too many tasks processing data blocks. Thus, *Ursa* generates larger network load than *CA*, which balances the computing workloads among servers. However, since *CA* does not aim to achieve long-term load balance, even though a server's average computing workload per second during T does not exceed its computing capacity, there may be some time epochs, during which its computing workload from concurrently submitted tasks exceeds its computing capacity. Therefore, *CA* generates larger network load than *CALV*, which balances the computing workloads over time in T among servers.

We then repeated the experiments with skewed data request distribution on blocks, since the workload usually are highly skewed to a few data blocks [11]. As [11], we set the number of task requests for each block follow a truncated power-law distribution with a lower bound and shape as 1 and 2, respectively. Figures 5(a) and 5(b) show the percentage of the network load of each load balancing method compared to the network load of *Random* with the *FIFO* and *Fair* schedulers, respectively under skewed data requests. They show that the network load follows 100%=*Random*>*Sierra*>*Ursa*>*CA*>*CALV* due to the same reasons as in Figures 4(a) and 4(b). Figures 5(a) and 5(b)

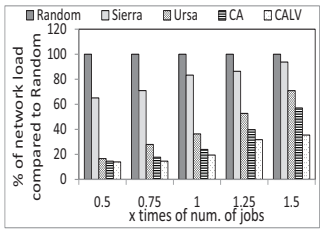


(a) Network load with FIFO

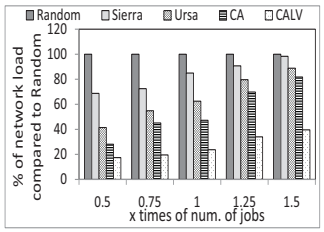


(b) Network load with Fair

Figure 4: Data locality performance of all methods

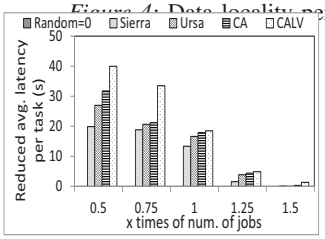


(a) Network load with FIFO

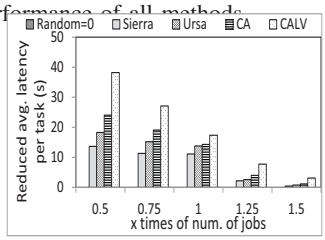


(b) Network load with Fair

Figure 5: Data locality performance with skewed data requests



(a) Random data request distribution



(b) Skewed data request distribution

Figure 6: Performance on task latency.

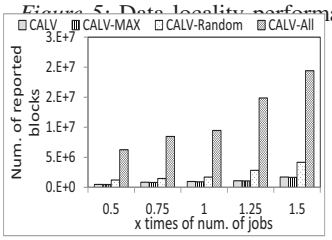


Figure 7: Performance on reducing network overhead.

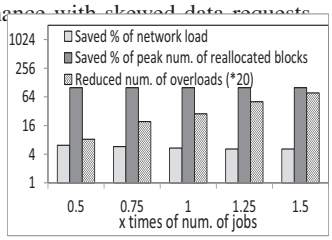


Figure 8: Effectiveness of lazy data block transmission.

indicate that *CALV* achieves higher data locality performance and hence saves much more network load than other methods with skewed requests.

2) Performance of Task Latency: In this section, we tested the performance of all load balancing methods with the *Delay* scheduler. We measured the task latency, which is the time between its submission until the end of its execution. A shorter task latency leads to a higher system throughput. Figures 6(a) and 6(b) show the reduced average latency per task of all methods compared to *Random*. They show that the result follows 0=*Random*<*Sierra*<*Ursa*<*CALV*. This is because if a method introduces more server overloads, tasks need to wait for a longer time until their data servers are available, which leads to longer average task latency. Therefore, these figures exhibit the opposite order of all methods compared to Figures 4 and 5. The figures indicate that *CALV* generates the shortest task latency among all the methods.

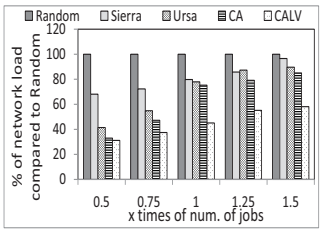
B. Cost-Efficiency of Load Balancing

To further reduce the computing and network loads of the load balancer and also reduce the network overhead in data reallocation, *CALV* has a coefficient-based data reallocation method to choose data blocks as few as possible to reallocate. In order to measure the effectiveness of this method, we measured the performance of *CALV* compared to *CALV-Max*, *CALV-Random* and *CALV-All*. In *CALV-Max*, each server reports the data blocks with the largest overload contributions until it is non-overloaded. In *CALV-Random*, each server randomly chooses the data blocks with positive overload contributions until it is non-overloaded. In *CALV-All*, each server reports all data blocks with positive overload contributions.

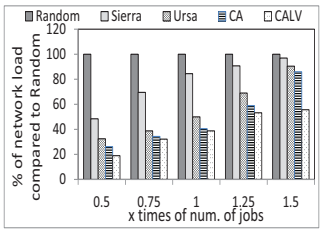
Figure 7 shows the number of blocks reported to the load balancer in all methods versus the number of jobs. The result follows *CALV-All*>*CALV-Random*>*CALV-Max*≈*CALV*. *CALV-Random* selects partial of all data blocks contributing workloads when the server is overloaded. Thus, it reports fewer data blocks than *CALV-All* which reports all such data blocks. *CALV-Max* and *CALV* report the data blocks with the largest overload contributions and load balancing coefficient, respectively, which contribute more computing workloads than other data blocks to server overloads. Therefore, *CALV* and *CALV-Max* report the smallest number of data blocks to the load balancer in all methods. The figure also shows that all methods report more data blocks when there are more jobs processed in the system. This is because more jobs lead to more server overloads, which produces more reported data blocks to the load balancer. This figure indicates that *CALV* and *CALV-Max* are effective in reducing the number of data blocks reported to the load balancer, leading to lower network load and computing overhead on the load balancer than other methods. However, since *CALV* chooses data blocks with the additional consideration of the second principle in Section III-C compared to *CALV-Max*, the computing capacities in source servers can be more fully utilized after data reallocation in *CALV* than in *CALV-Max*.

C. Performance of Lazy Data Transmission

In this section, we present the performance of *CALV*'s lazy data transmission method in reducing the peak network overhead for data reallocation and improving the load balance performance. Recall that this method can avoid overloading the destination servers. If a destination server is overloaded, the task whose data server is this destination server will be allocated to another server and its required block needs to be transmitted, which generates network load. Figure 8 shows the saved percentage of network load calculated by $(nl-nl')/nl$, where nl and nl' are the network loads of *CALV* without and with the lazy transmission method, respectively. It shows that the lazy transmission method can save at least 5.1% network load. Without this method, the destination server may be overloaded earlier



(a) Network load with FIFO



(b) Network load with Fair

Figure 9: Data locality performance on a Hadoop cluster.

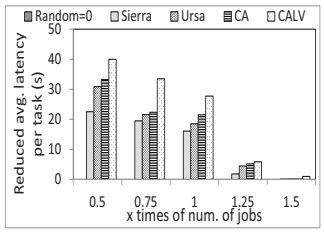


Figure 10: Reduced task latency on a Hadoop cluster.

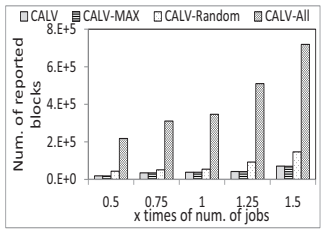


Figure 11: Num. of reported blocks on a Hadoop cluster.

than the source server due to hosting the migrated block. With lazy transmission, the network load due to such kind of overloads in the destination servers can be avoided. This is confirmed by the result of the reduced number of overloads in Figure 8. It shows that the reduced number of overloads increases when the number of jobs increases.

Figure 8 also shows the saved percentage of the peak number of reallocated blocks during a second within T of *CALV* with the lazy data block transmission method compared to *CALV* without this method. We see that this method saves at least 99.95% of the peak number of data blocks reallocated. Without this method, all data blocks are reallocated right after the reallocation scheduling. By arranging the data transmissions at different times, the lazy data block transmission method releases the peak network overhead in data reallocation. This figure verifies that the lazy data transmission method can reduce the peak network overhead for data reallocation and improve the load balance performance.

V. Performance Evaluation on a Real Cluster

In this section, we present the experimental results on a real cluster. We implemented *CALV* and its comparison methods on the Apache Hadoop (version 1.2.1) on Palmetto [21], which is a computing cluster consisting of 771 8-core nodes. Due to the limitation of usage, we randomly selected 100 nodes to form a computing cluster. Since the server scale is reduced to $1/30$ as in the trace, we reduced the number of total jobs to $1/30$ of the number of jobs in the trace. Also, due to the storage usage limitation on each node, we set each server's storage capacity and the input/output size of a job to be $1/1000$ of their original settings. All other settings remain the same as in simulation. We measured the performance of *CALV* with the *FIFO*, *Fair* and *Delay* schedulers, respectively, by repeating the experiments in Sections IV-A1 and IV-A2.

Figures 9(a) and 9(b) show the percentage of the network load compared to *Random* of all methods versus the number of jobs using the *FIFO* and *Fair* schedulers, respectively. They illustrate the same order and trend of all methods as in Figures 4(a) and 4(b), respectively, due to the same reasons. The results confirm that *CALV* can save more network load than all other methods with its computing load aware load balancing on a real computing cluster.

Figure 10 shows the reduced task latency of all methods compared to *Random*. It demonstrates the same order and trend of all methods as in Figure 6(a) due to the same reasons. It confirms that *CALV* can reduce the task latency and improve the task throughput with the *Delay* scheduler on a real computing cluster.

Figure 11 shows the number of blocks reported to the load balancer in all methods. It illustrates the same order and trend of all methods as in Figure 7 due to the same reasons. It indicates that the coefficient-based data reallocation in *CALV* is effective in reducing the computing and network overhead of the load balancer and the network overhead in data reallocation.

VI. Related Work

Many research efforts have been devoted to data allocation in large-scale computing clusters. They either randomly allocate data without the load balance consideration or balance the number of data blocks or the I/O load in servers.

Random data allocation. The works in [2, 6, 27] randomly allocate data blocks to servers in the cluster in order to balance the storage load. Weil *et al.* [7] proposed to randomly select data blocks to be reallocated to the newly added server in order to balance the storage utilization between the existing servers and newly added servers, and randomly allocate data blocks stored in a failed server to all other servers to maintain the load balance among servers. However, all these data allocation methods cannot avoid server overloads due to computing workloads.

Balancing the number of data blocks. In [8], the servers are divided into two groups, primary and secondary servers, and the primary replica and secondary replicas of each block are stored in these two groups accordingly. Within each group, the data blocks are evenly distributed among servers, and the requests of a data block are evenly distributed between the primary replica and secondary replicas. Hsiao *et al.* [9] assumed that the computing workloads in servers are proportional to the number of blocks stored in them and aimed to balance the number of data blocks among servers. With the same assumption, Thereska *et al.* [10] proposed to uniformly allocate data blocks to all servers. However, this assumption does not hold true in reality. Our analysis on the real trace in Section II-B demonstrates the high diversity of computing workloads targeting different data blocks. Therefore, these methods cannot avoid the server overloads due to computing workloads by allocating data

blocks without considering the difference of their associated computing workloads.

Balancing the I/O load. You *et al.* [11] found that the I/O workload varies among data blocks in servers, and there are servers over-utilizing their I/O capacities. Thus, they proposed *Ursa* to migrate data blocks in these servers to servers not fully utilizing their I/O capacities, with bandwidth cost minimization. Bonvin *et al.* [12] proposed a self-managed key-value storage service in cloud storage. Each data partition migrates or replicates itself by considering both monetary payment to cloud providers and its popularity in order to balance the queries among servers and meanwhile minimize the payment cost. In [13, 14], the problem of data allocation with I/O load balance and reallocation cost minimization is proved to be NP-Hard, and heuristic solutions are proposed for this problem.

CALV is the first work that balance the computing workloads with the consideration of the differences of computing workloads associated with data blocks in servers. It is also novel in that it achieves load balance in a long term rather than at a time spot. *CALV* also improves the cost-efficiency and scalability while achieving its long-term load balance goal.

VII. Conclusion

Through an analysis of a Facebook cluster's job running trace, we show the importance of considering the computing workloads in load balancing. We then propose a Computing load Aware and Long-View load balancing method (*CALV*). *CALV* is cost-efficient and creative in two features: i) it considers computing workload in load balancing, and ii) it achieves long term load balance. To achieve these objectives, when selecting data blocks to migrate out from an overloaded server, *CALV* chooses the blocks that contribute more workloads at the server's more overloaded epochs and contribute less workloads at the server's more underloaded epochs. To further improve the load balance performance, *CALV* incorporates a lazy data block transmission method. It chooses a time for each data migration in order to reduce the destination server overloads, and release the peak network overhead for data reallocation. The trace-driven experiments on both simulation and a real computing cluster show that *CALV* outperforms other methods in improving data locality, reducing task delay, network load and reallocation overhead. In the future, we will study the dynamic load balancing within T for jobs without planned submission times.

Acknowledgements

This research was supported in part by U.S. NSF grants NSF-1404981, IIS-1354123, CNS-1254006, CNS-1249603, Microsoft Research Faculty Fellowship 8300751.

References

[1] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of SIGOPS*, 2003.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of MSST*, 2010.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[4] 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 *Proc. of EuroSys*, 2010.

[5] Apache. Fair Scheduler. http://hadoop.apache.org/docs/r1.2.1 /fair_scheduler.html, 2010, [accessed in July 2015].

[6] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proc. of SIGMOD*, 2011.

[7] S. A. Weil, S. A. Brand, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proc. of OSDI*, 2006.

[8] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and Flexible Power-Proportional Storage. In *Proc. of SoCC*, 2010.

[9] H. Hsiao, H. Su, H. Shen, and Y. Chao. Load Rebalancing for Distributed File Systems in Clouds. *TPDS*, 2013.

[10] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proc. of EuroSys*, 2011.

[11] G. You, S. Hwang, and N. Jain. Scalable Load Balancing in Cluster Storage Systems. In *Proc. of Middleware*, 2011.

[12] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage. In *Proc. of SoCC*, 2010.

[13] D. Kunkle and J. Schindler. A Load Balancing Framework for Clustered Storage Systems. In *Proc. of HiPC*, 2008.

[14] E. A. Eric, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly Finding Near-Optimal Storage Designs. *TOCS*, 2005.

[15] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proc. of MASCOTS*, 2011.

[16] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. of EuroSys*, 2013.

[17] A. Verma, L. Cherkasova, and R. H. Campbell. RIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *Proc. of ICAC*, 2011.

[18] I. Gupta, B. Cho, M. R. Rahman, T. Chajed, N. Abad, C. L.and Roberts, and P. Lin. Natjam: Eviction Policies For Supporting Priorities and Deadlines in Mapreduce Clusters. In *Proc. of SoCC*, 2013.

[19] A. Beloglazov and R. Buyya. Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. *CCPE*, 2011.

[20] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *Proc. of INFOCOM*, 2012.

[21] Palmetto Cluster. http://http://citi.clemson.edu/palmetto/, [accessed in July 2015].

[22] P. Vagata and K. Wilfong. Scaling the Facebook data warehouse to 300 PB. https://code.facebook.com/posts/22986 1827208629/scaling-the-facebook-data-warehouse-to-300-pb/, [accessed in July 2015].

[23] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of SIGCOMM*, 2008.

[24] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. In *Proc. of NSDI*, 2013.

[25] Apache Hadoop FileSystem and its Usage in Facebook. http://cloud.berkeley.edu/data/hdfs.pdf.

[26] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs Scale-out for Hadoop: Time to rethink? In *Proc. of SoCC*, 2013.

[27] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. Kingfisher: Cost-aware elasticity in the cloud. In *Proc. of INFOCOM*, 2011.