This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

Towards Long-View Computing Load Balancing in Cluster Storage Systems

Guoxin Liu, *Student Member IEEE*, Haiying Shen*, *Senior Member IEEE*, Haoyu Wang *Student Member IEEE*

Abstract—In large-scale computing clusters, when the server storing a task's requested data does not have sufficient computing capacity for the task, current job schedulers either schedule the task to the closest server and transmit to it the requested data, or let the task wait until the server has sufficient computing capacity. The former solution generates network load while the latter solution increases 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 do not aim to balance the computing load for the long term. 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*). *CALV* is novel in that it achieves long-term computing load balance by migrating out an overloaded server data blocks contributing 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. Based upon the task schedules, we further propose a task reassignment algorithm that reassigns tasks from an overloaded server to other data servers of the tasks to make it non-overloaded before *CALV* is conducted. The above methods are for the tasks whose submission times and execution latencies can be predicted. To handle unexpected tasks or insufficiently accurate predictions, we propose a dynamic load balancing method, in which an overloaded server dynamically redirects tasks to other data servers of the tasks, or replicates the tasks' requested data to other servers and redirects the tasks to those servers in order to become non-overloaded. Finally, we propose a proximity-aware tree based distributed load balancing method to reduce the reallocation cost and improve the scalability of *CALV*. 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.

1 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 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 from unpredictable performance variations [3] due to the resource multiplexing between them.

Current computing clusters depend on job schedulers [2, 5, 4] to improve system efficiency such as data locality and task delay. Preserving "data locality" means putting computing workloads, such as mapper tasks in MapReduce [3], with their requested data. When a task's data servers (i.e., the servers storing the task's requested 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, this method cannot preserve data locality because it requires data transmission from the task's data server to its allocated server. In order to preserve the data locality, the Delay scheduler [4] postpones running a task until its data server has sufficient computing capacity, which generates an extra delay for task execution. Therefore, job schedulers cannot preserve data locality exclusively without causing 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. 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 these data blocks. However, most previously proposed load balancing methods [2, 6–16] for cluster storage systems do not consider the computing workload, which is the bottleneck in data-intensive applications. Though the CDRM [17] load balancing method considers computing workload, it does not balance the computing load for the long term. It aims to achieve the load balanced state at the time of executing the load balancing method rather than for the long term. As a result, a server may still become overloaded before the next load balancing execution. One solution for this problem is to set a very small time interval for the periodical load balancing execution, which leads to very high overhead. Therefore, it is critical to balance the computing workloads for the long term.

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

- * *Corresponding Author. Email: hs6ms@virginia.edu; Phone: (434) 924-8271; Fax: (434) 982-2214.*
- *Haiying Shen and Haoyu Wang are with the Department of Computer Science, University of Virginia, Charlottesville, VA, 22904. E-mail: {hs6ms, hw8c}@virginia.edu, {guoxinl}@clemson.edu*

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

cluster today [18], and the scale increases rapidly over time. A load balancer needs to keep track of the workload for each data block [11, 13] and schedules "data reallocation" among servers. Also, migrating many data blocks within a short time period in reallocation generates a high network peak load in 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 [18, 2, 19], which trace is a typical MapReduce trace with many jobs concurrently running on a large storage cluster. It is widely used to analyze the clusters computing workloads [20–23], which shows that i) the computing workloads of tasks are heterogenous, ii) there are a large number of server overloads due to insufficient computing capacities, and iii) the server overloads either break 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 is designed for a storage cluster with jobs concurrently running on it for a long time, so that the data locality and task delay are important issues for tasks competing resources mutually. It consists of the following methods.

Coefficient-based data reallocation. An overloaded server is overloaded at some epochs but may be underloaded at some 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 reallocated. Specifically, higher coefficients are assigned to data blocks 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.

Lazy data block transmission. By selecting a time to migrate a data block from source server to the destination server in the data allocation schedule, this method avoids high network peak load and improves the load balanced state.

Enhancement methods for CALV. *Task reassignment algorithm;* It reassigns tasks from an overloaded server to other data servers of the tasks to make it non-overloaded before *CALV* is conducted. *Dynamic load balancing method;* An overloaded server dynamically reallocates tasks to other data servers of the tasks, or replicates the tasks' requested data to other servers and redirects the tasks to those servers in order to become non-overloaded. *Proximity-aware tree based distributed load balancing.* It builds a tree by connecting proximity-close servers and conducts data block migrations between proximity-close servers in the bottom-up manner along the tree in order to reduce the reallocation cost and improve the scalability of *CALV*.

The coefficient-based data reallocation method are for the tasks whose submission times and execution latencies can be pre-known. The dynamic load balancing method handles unexpected tasks or insufficiently accurate predictions.

TABLE 1: Notations.

s_i	server i	$C^c_{s_i}$	s_i's computing capacity
D_{s_i}	the set of data blocks stored in s_i	d_j	data block j
t_i	task i	e_k	time epoch k in T
$u^{s_i}_{e_k}$	s_i's unbalanced workload at e_k	$C^c_{t_i}$	t_i's required amount of computing resource
$e^s_{t_i}$	t_i's submission time	$e^f_{t_i}$	t_i's finishing time
$L^{d_j}_{e_k}$	computing workloads targeting d_j during e_k	$L^{s_i}_{e_k}$	computing workloads on s_i during e_k

Trace-driven experiments in simulation and a real cluster show the effectiveness of *CALV* in balancing the computing workloads and its high cost-efficiency. *CALV* is the first work that balances the computing workloads for the long term among servers by reallocating data blocks among them. It is suitable in a scenario in which a large number of jobs are submitted periodically [24]. The rest of this paper is organized as follows. Section 2 presents the preliminaries of the load balancing problem and the trace analysis results. Section 3 presents the design of *CALV* and its enhancement methods in detail. Sections 4 and 5 present the performance evaluation of *CALV* in simulation and a real cluster. Section 6 presents the related work. Section 7 concludes this paper with remarks on our future work.

2 Computing Load Aware Load Balancing Problem

2.1 Preliminaries

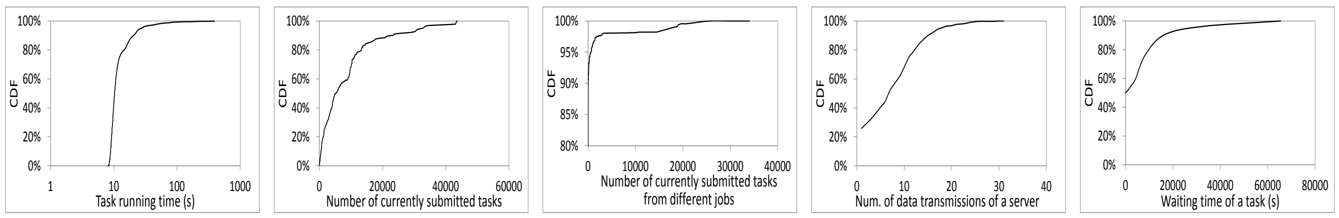
In this section, we present the environment of the cluster storage system and its load balancing problem. Table 1 lists the important notations used in this paper. In a cluster storage system, we use S to denote the set of all servers, and s_i to denote the i^{th} server.

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 [2, 11, 8]. We use d_j to denote the j^{th} data block. 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 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 [25, 26] on the same type of files, 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 the same type of data for advertisement, spam detection

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713



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

Fig. 1: Trace data analysis results.

and so on [27]. For a new submitted job without an execution historical log or a previous job on a different type of data, we can get its process execution time by profiling run [25, 28]. We also assume that each task's submission time is predictable in advance according to its historical running records [25] 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.

The data can be short-lived and long-lived [14, 16, 29] and the predicted execution time by tracking may not be accurate especially for short-lived data blocks. Also, the submission times of some jobs may not be pre-known or the predicted submission times may not be accurate. To handle these problems, we propose a dynamic load balancing method (Section 3.6).

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 which does not have enough computing resource for the task, we call this server 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 [30, 31].

2.2 Trace Data Analysis

In this section, we analyze a Facebook Hadoop cluster trace [18, 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 number of tasks of a job can be calculated by dividing its input size by block size. To determine the type of each job, we randomly selected a job testing example in Apache Hadoop. The Hadoop cluster consists of 8 nodes in Palmetto [32], each of which has 8 cores and 32GB memory. 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. The running time of 56.5% tasks are longer than 10s. 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 greatly 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. The longer tasks dominant the workload and we need to balance the workload with a long term view.

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 with 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. If there are no concurrently submitted jobs, balancing the data blocks processed by a job among servers may solve the problem. However, a data block may still be requested by many different jobs submitted at different time, therefore, we still need to achieve a data allocation with data block balancing among servers for all jobs. Otherwise, data blocks may still be transferred among servers to achieve the data locks balancing for next job. Therefore, we still need a load balancer with long-view computing load balancing to improve the data locality. 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 s_i does not have sufficient computing capacity for a task, the job schedulers handle this case in two ways. The *FIFO* scheduler [2] allocates the task to the closest server that has sufficient computing capacity and the data is transmitted from s_i to this server, which generates network load. The *Delay* [4] scheduler lets the task wait until s_i has sufficient computing capacity, which 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 [18] with 8 computing slots in each server and 10PB of data [33] randomly distributed among all servers. We simulated the 24442 jobs [18] 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

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

of data transmissions from a server when it is overloaded with the *FIFO* scheduler. We see that more than 50% of all servers transmit more than 6 data blocks to other servers. It indicates that the data locality preservation is worsened due to the server overloads. Figure 1(e) shows the CDF of the waiting time of all tasks. We see that around 50% of all tasks wait 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 for improving data locality and reducing task latency.

3 Computing Load Aware and Long-View Load Balancing

3.1 System Overview of *CALV*

Motivated by our trace study, we propose a Computing load Aware Long-View load balancing method (*CALV*) with high cost-efficiency in a cluster storage system. Based on the computing workloads targeting its data blocks at each epoch (e.g., the number of computing slots in each second), each server checks if it will become overloaded in the next T. Each overloaded server selects data blocks to migrate out to release its excess computing workloads while fully utilizing its computing capacity over T. It reports the workloads of these blocks to the load balancer. Each server also 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 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 to 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 a lazy data block transmission method to improve the load balance performance and reduce network peak load. Since the source server and destination server of a migration block may be overloaded at some epochs while non-overloaded at other epochs, 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 this data block. This way, we can try to avoid the situation that the destination becomes overloaded by hosting this data block.

3.2 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 ($e^s_{t_i} \leq e_k \leq e^f_{t_i}$). Note that each data block may be updated periodically. 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 by:

$$L^{d_j}_{e_k} = \sum_{t_i \in T^{e_k}_{d_j}} C^c_{t_i} \tag{1}$$

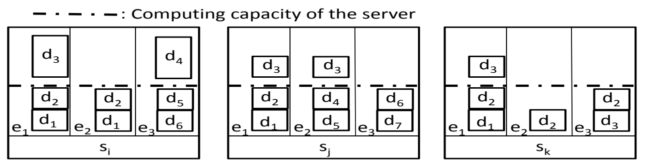
Recall that the whole set of data blocks stored in server s_i is denoted by D_{s_i}. Then, we can get the workloads on s_i during epoch e_k by:

$$L^{s_i}_{e_k} = \sum_{d_j \in D_{s_i}} L^{d_j}_{e_k} \tag{2}$$

For each server s_i, at epoch $e_k \in T$, if $L^{s_i}_{e_k} > C^c_{s_i}$, 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 excess computing workload and report the workload of each of these blocks at each epoch as $L^{d_j}_{e_1}, L^{d_j}_{e_2}, ..., L^{d_j}_{e_n}$. The pre-selection increases the scalability of the load balancer. If the load balancer considers all the data blocks to achieve load balance, it cannot be scalable since there are a large amount of data blocks in the system.

3.3 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.



(a) Reduce num. of reported data blocks in spatial space (b) Reduce num. of reported data blocks in temporal space (c) Avoid server underload

Fig. 2: Selection of data block to reallocate.

Rationale of the migration data block selection policy. We use an example shown in Figure 2 to illustrate the rationale of our migration data block selection policy. In the figure, the height of each data block represents the computing workload targeting this data block during an epoch. For example, in Figure 2(a), the workloads of data blocks 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 equals to 2 computing slots ($L^{d_3}_{e_1} = 2$). When an overloaded server selects the data blocks to migrate out, it follows two principles as explained below.

Each overloaded server should try to reduce the number of selected data blocks. It not only reduces the size of information reported to the load balancer hence its network

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

load and computing load but also 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 an 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 of 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 excess workload in both e_1 and e_2. Selecting any other block in e_1 and e_2 can only release the excess 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 excess load in an overloaded epoch, a server may become more underloaded at other epochs. For example, as shown in Figure 2(c), though reallocating d_2 or d_3 releases the excess 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 contributing 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

$$u^{s_i}_{e_k} = \begin{cases} L^{s_i}_{e_k} - C^c_{s_i} & \text{if } L^{s_i}_{e_k} \neq C^c_{s_i} \\ -c & \text{otherwise} \end{cases} \tag{3}$$

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:

$$\sum_{\forall e_k \in T} u^{s_i}_{e_k} \cdot L^{d_j}_{e_k} \tag{4}$$

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

For an overloaded server, releasing its excess 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 d_j to the overload of an overloaded server s_i. It is calculated by $\sum_{\forall e_k \in T} o^{s_i}_{e_k} \cdot L^{d_j}_{e_k}$, where $o^{s_i}_{e_k} = L^{s_i}_{e_k} - C^c_{s_i}$ if $L^{s_i}_{e_k} > C^c_{s_i}$ and $o^{s_i}_{e_k} = 0$ otherwise.

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 their coefficients. 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 $L^{s_i}_{e_k} \leftarrow L^{s_i}_{e_k} - L^{d_j}_{e_k}$. 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 of 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 $L^{d_j}_{e_1}, L^{d_j}_{e_2}, ..., L^{d_j}_{e_n}$.

Data reallocation scheduling at the load balancer. Each server reports its computing capacity ($C^c_{s_i}$) and computing workload at each epoch e_k to the load balancer ($L^{s_i}_{e_1}, L^{s_i}_{e_2}, ..., L^{s_i}_{e_n}$). The load balancer sorts the servers in the descending order of $\sum_{\forall e_k \in T}(C^c_{s_i} - L^{s_i}_{e_k})$. It 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 since overloaded servers in our method may have underloaded epochs in T before and during the reallocation scheduling.

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., $\sum_{\forall e_k \in T} L^{d_j}_{e_k}$ and the underloaded servers are ordered in descending order of their accumulated available capacity during T, i.e., $\sum_{\forall e_k \in T}(C^c_{s_i} - L^{s_i}_{e_k})$. Starting from the first data block d_i in the sorted block list, the load balancer reallocates it from its source server s_i to another server. 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 $L^{s_j}_{e_k} \leftarrow L^{s_j}_{e_k} + L^{d_i}_{e_k}$. 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. It then chooses the server with the smallest overload contribution as block d_i's

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

reallocated server. In 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.

3.4 Lazy Data Block Transmission

The coefficient-based data reallocation method generates a new data allocation schedule offline. Reallocating the data blocks right after the reallocation scheduling may generate tremendous network loads in a short time and also overload the destination server, which may delay 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 to release the overload in s_i. 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, both s_i and s_j will not be overloaded. Then, the load balancer 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.

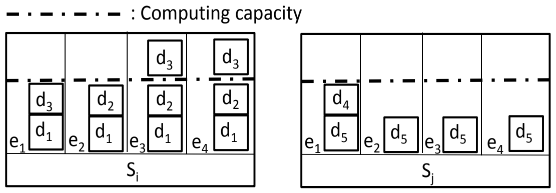


Fig. 3: Lazy data block transmission.

Below, we introduce how the load balancer determines the block transmission time for each data block to be reallocated in order to avoid overloading the destination server and reduce the peak network load. For data reallocation of each data block, say d_k, from server 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 $e^o_{s_i,d_k}$ and $e^o_{s_j,d_k}$, respectively. If $e^o_{s_i,d_k} > e^o_{s_j,d_k}$, 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 $e^o_{s_i,d_k}$, denoted by $e^f_{s_j,d_k}$. Then, the load balancer randomly selects an epoch within $(e^f_{s_j,d_k}, e^o_{s_i,d_k})$ for d_k's reallocation. Take d_3 in Figure 3 for example, we can get $e^o_{s_i,d_3} = e_3$, $e^o_{s_j,d_3} = e_1$. Since $e^o_{s_i,d_3} > e^o_{s_j,d_3}$ and $e^f_{s_j,d_k} = e_1$, the data can be transmitted within (e_1, e_3), which is e_2. If $e^o_{s_i,d_k} \leq e^o_{s_j,d_k}$, 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.

3.5 Task Reassignment Algorithm

The basic *CALV* method moves blocks from predicted overloaded servers to underloaded servers. Recall that each block has several replicas. Then, rather than relying on data reallocation, a task assigned to a predicted overloaded server can be reassigned to an underloaded data server of the task (which has a replica of the task's requested data block). In this way, overloaded servers can be avoided without the need of data reallocation, which saves network overhead. For example, in Figure 4, the tasks requesting block d_3, d_4 and d_5 originally assigned to server s_i are reassigned to server s_j. Then, s_i will not be overloaded and the resources in s_j are fully utilized. Accordingly, we propose task reassignment method (*TR*) to complement *CALV*. *TR* requires the cooperation between the job scheduler and the load balancer. That is, the job scheduler needs to predetermine the allocated servers for tasks, and then notifies the servers of their assigned tasks. Each server predicts its overload status. If it predicts that it will be overloaded, it reports to the load balancer its assigned tasks and their requested data blocks in the form of $< t_i, b_i >$. The load balancer has a view of the data allocation among the servers. Then, the load balancer can reassign tasks on overloaded servers to the tasks' underloaded data servers. After the task reassignment, it conducts *CALV* to achieve load balance by data reallocation.

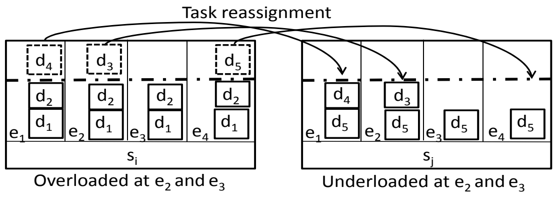


Fig. 4: Task reassignment.

During the task reassignment process, for faster load balance convergence, the load balancer gives more overloaded servers (measured by $\sum_{\forall e_k \in T}(L^{s_i}_{e_k} - C^c_{s_i})$) a higher priority to reassign their tasks out. Also, we aim to move out as few tasks as possible from an overloaded server s_i to make it non-overloaded. Below, we introduce how to find tasks to reassign based on this principle. First, we find the epochs that server s_i is overloaded, combine consecutive epochs and generate overloaded epoch periods such as $T_e = \{e_1, e_2 - e_3, e_4\}$. For a single epoch e_k, such as e_1 and e_4, among the tasks generating workload in epoch e_k, the task with workload no less than and closest to $(L^{s_i}_{e_k} - C^c_{s_i})$ has the highest priority to be reassigned. If such a task does not exist, we order these tasks in descending order of their workloads and select the tasks in the top-down manner to reassign until the server is non-overloaded in epoch e_k. In this way, the excess load can be released quickly and the resources can be more fully utilized. The procedure to handle a single epoch actually can be generalized to the procedure of handling combined consecutive epochs introduced below.

For epoch period $e_i - e_j$ (e.g., $e_2 - e_3$), we first find tasks that maximally cover consecutive epochs during $e_i - e_j$ but do not cover the epochs before e_i and after e_j. The first condition aims to reduce the number of tasks for reassignment. The latter condition aims to avoid the transfer of tasks between servers during their running time. Therefore, we group tasks that cover the same epoch period $T'_e \in T_e$, and rank the task groups in descending order of the number of their covered epochs. We pick each task group in the top-down manner and select tasks to reassign to remove excess load of server s_i in this group's T'_e. After we remove excess load of each epoch period $T'_e \in T_e$, server s_i becomes non-overloaded. We use $C^c_{t_i,e_k}$ to denote the required amount of computing resource of t_i at epoch e_k. To choose tasks in a group to resign, the task with $C^c_{t_i,e_k} \geq (L^{s_i}_{e_k} - C^c_{s_i})$ and $\min\{\sum_{e_k \in T'_e}\{C^c_{t_i,e_k} - (L^{s_i}_{e_k} - C^c_{s_i})\}\}$ for each epoch $e_k \in T'_e$ has the highest priority to be reassigned to release

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

all the excess load in T'_e of server s_i. If there is no such a task, the tasks in the group are ordered in descending order of workload and are selected in sequence until s_i is not overloaded during T'_e. Reassigning these selected tasks can release excess load in each overloaded epoch maximally while fully utilizing the resources in server s_i during T'_e.

Then, the selected tasks are reassigned to underloaded data servers of the tasks during their epoch periods. We use $T_e^{t_i}$ to denote the covered epoch period of task t_i. Task t_i that targets data block d_j during epoch $T_e^{t_i}$ is reassigned to another server that hosts d_j and has sufficient computing capacity for the task during each epoch $e_k \in T_e^{t_i}$. Therefore, we first find the servers that satisfy $(C^c_{s_i} - L^{s_i}_{e_k}) \geq C^c_{t_i,e_k}$ for each epoch $e_k \in T_e^{t_i}$. Among these servers, the one with $\min\{\sum_{e_k \in T_e^{t_i}} (C^c_{s_i} - L^{s_i}_{e_k}) - C^c_{t_i,e_k}\}$ is selected to reassign task t_i. In this way, the selected server will not be overloaded by hosting task t_i while the resources can be fully utilized.

3.6 Dynamic Load Balancing Method

Our proposed methods in the above assume that tasks' submission times are predictable or pre-indicated. However, some tasks in the next time interval T may not be predictable or their submission times are only indicated a few epochs in advance. In this case, even after the task reassignment algorithm and the *CALV* algorithm are executed offline to reach load balance in the next time interval T, some servers may still become overloaded during the next T. To avoid this problem, we propose a dynamic load balancing method (*DLB*) that is executed during task running time in interval T

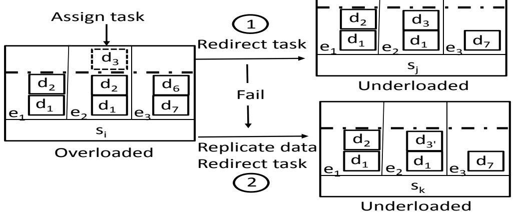


Fig. 5: Dynamic Distributed Task Assignment Method

As shown in Figure 5, when a server is overloaded, it first redirects tasks to other data servers of the tasks that have sufficient available computing capacity. If it cannot find such data servers, it replicates the tasks' requested data blocks to other servers that have sufficient available computing capacity and then redirects the tasks to these servers. In this example, server s_i becomes overloaded in epoch e_2. Then, since server s_j hosts d_3 and has available computing capacity for the task at epoch e_2, the task targeting d_3 can be redirected to server s_j at epoch e_2. If there is no data server (which hosts d_3) of the task that has sufficient available computing capacity at epoch e_2, then a replica of block d_3 is created ($d_{3'}$ in the figure) at server s_k that has available computing capacity at epoch e_2 for the task, and the task is then redirected to s_k.

We can rely on the load balancer to conduct *DLB*. That is, each overloaded server reports its assigned tasks and their targeting blocks to the load balancer (i.e., $< t_i, b_i >$), and the load balancer then conducts the *DLB* algorithm. After receiving the notification from the load balancer, the overloaded server redirects task directly or first replicates data and then redirects task. In order to reduce the load on the centralized load balancer, the direct task redirection algorithm can be executed in a distributed manner. That is, after executing *CALV*, the load balancer notifies each server other replica servers of each of its blocks. Then, when a server becomes overloaded, it contacts the replica servers of its blocks to reassign its tasks. The distributed method is not suitable for task redirection that needs data replication because it requires each node to always have an updated global knowledge of the workload schedule in each epoch in each server.

Each server runs the *DLB* algorithm when it becomes overloaded. Since the overload is caused by unexpected tasks, overloaded server s_i tries to redirect the unexpected tasks to other servers in order to keep the original task assignment schedule as much as possible. Server s_i ranks the unexpected tasks in descending order of their workloads and reassigns the tasks in the top-down manner. For a picked task t_i, it asks each data server of t_i whether it has sufficient available computing capacity to host t_i, i.e., satisfies $(C^c_{s_j} - L^{s_j}_{e_k}) \geq C^c_{t_i,e_k}$ for each epoch $e_k \in T_e^{t_i}$. If server s_j receives positive responses from several servers, it redirects task t_i to the server with $\min\{\sum_{e_k \in T_e^{t_i}} (C^c_{s_j} - L^{s_j}_{e_k}) - C^c_{t_i,e_k}\}$. In this way, the selected server has sufficient computing capacity to host task t_i while its resources are more fully utilized. If server s_i becomes non-overloaded after the task redirection, the *DLB* process stops. Otherwise, the next task is picked and the same process is conducted. After all the tasks are checked, server s_i conducts data replication and task redirection.

In the data replication procedure, server s_k checks the remaining unexpected tasks. It reports the first task $t_i =< e^s_{t_i}, e^f_{t_i}, C^c_{t_i} >$ to the load balancer. Among the servers that do not have task t_i's requested block b_i, the load balancer finds the servers that have sufficient computing capacity to host t_i (i.e., satisfy $(C^c_{s_j} - L^{s_j}_{e_k}) \geq C^c_{t_i,e_k}$ for each epoch $e_k \in T_e^{t_i}$). It then selects the server with $\min\{\sum_{e_k \in T_e^{t_i}} (C^c_{s_j} - L^{s_j}_{e_k}) - C^c_{t_i,e_k}\}$ and notifies server s_i of this selected server s_j. Next, overloaded server s_i replicates task t_i's requested data block to s_j and redirects t_i to s_j. Then, server s_i checks if it becomes non-overloaded. If yes, it stops redirecting tasks.

3.7 Proximity-Aware Tree based Distributed Load Balancing

The network load increases when either the size of transmitted data or the transmission path length increases [30, 31]. Therefore, besides reducing the number of data blocks in reallocation, it is also important to reduce the path length of data block transmission. Thus, *CALV* tries to find a destination server physically close to a source server for data reallocation. Also, it is important to conduct the load balancing in a distributed manner in order to release the computing and network overloads on the load balancer. For these two purposes, *CALV* builds a proximity-aware tree based load balancing infrastructure as shown in Figure 6. The load balancer knows the topology of the cluster such as the fat-tree infrastructure [34]. Also, all servers report their computing capacities to the load balancer. The load balancer builds a m-nary proximity-aware tree, in which each root of a sub-tree has m number of children and connected servers are physically close to each other. First, the load balancer builds a tree by temporarily regarding one of the

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

core routers as the root and all servers as leaves. Next, to create each sub-tree, the load balancer selects the server with the largest computing capacity among all m servers inside the sub-tree to be the root of this sub-tree. The sub-trees are built in the bottom-up manner and finally the load balancer itself becomes the root of the entire tree. After the tree is built, the load balancer notifies all servers of their parents and children in the tree.

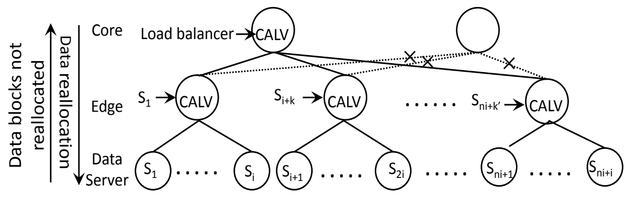


Fig. 6: Proximity-aware tree based distributed load balancing.

In load balancing, each server reports information as indicated in Section 3.2 to its sub-tree root, which conducts data reallocation as described in Sections 3.3 and 3.4. Different from the data reallocation scheduling explained previously, one data block d_j is only allocated to a destination server s_i with 0 overload contribution (a server is overloaded due to the workload contributed by d_j) from d_j, i.e., s_i is not overloaded in any epoch with computing workload towards d_j. This is because there may be a server in other sub-trees that has 0 overload contribution from d_j.

After the reallocation scheduling, a root of a sub-tree may not be able to find destination servers to reallocate some reported data blocks. Then, the root reports the information of these data blocks and all servers in its sub-tree to its parent, which will schedule the reallocation for these blocks. Thus, the load balancing is conducted in the bottom-up manner and finally the information of unsolved blocks and all servers are reported to the load balancer. The load balancer itself conducts the data reallocation as explained previously. As a result, the load balancing is conducted in a distributed manner and the data is transmitted between physically close servers in data reallocation. The CALV runs periodically so that the proximity-aware tree can be reconstructed every time. If the closest server fails, the overloaded server will continue searching for secondary closest node.

4 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 [18, 2]. Based on the trace, we simulated 3000 servers in a computing cluster with the typical fat-tree topology [34]. 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. As [25, 28], we set each task's execution time as its execution time in the profiling running in Palmetto in Section 2.2. The storage and computing capacities of each server were set to 12TB [35, 36] and 8 computing slots [32, 37], respectively. The default size of a block and the number of replicas were set to 128MB 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 [33]. By default, the requested data block of each task was randomly chosen from all blocks. To randomly choose a data block from M (80M) blocks, we generated a random number in the range of $[1, M]$, denoted by m. Then, the m^{th} data block is the randomly chosen block. We set the load balancing execution 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]. *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. We created a Computing load Aware load balancing method (CA) for comparison. CA uses the average computing workload per second during load balancing time period to measure the block load in load balancing, and aims to balance the average computing workload among servers. It can represent a modified current computing aware load balancing method [17]. 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.

4.1 Effectiveness of Load Balancing

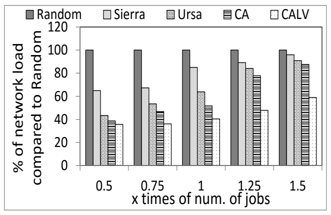
4.1.1 Performance of Data Locality with the FIFO and Fair Schedulers

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 [30, 31].

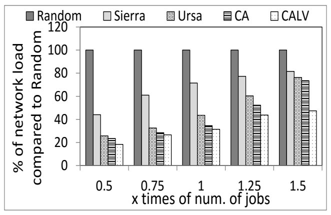
Figures 7(a) and 7(b) show the percentage of the network load of each load balancing method compared to the network load of *Random* using the *FIFO* and *Fair* schedulers, respectively. The result follows 100%=*Random*>*Sierra*>*Ursa*>*CA*>*CALV*. *Random* allocates data blocks to all servers randomly 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. *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

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

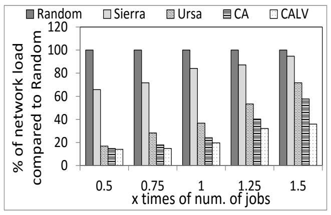


(a) Network load with FIFO

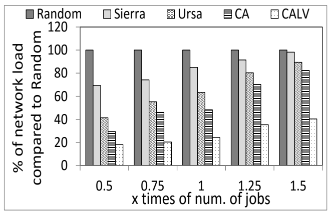


(b) Network load with Fair

Fig. 7: Data locality performance of all methods.

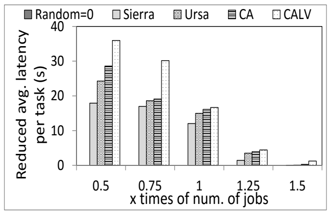


(a) Network load with FIFO

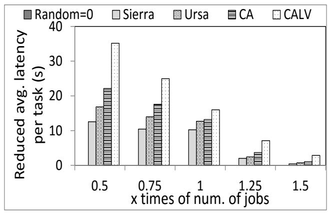


(b) Network load with Fair

Fig. 8: Data locality performance with skewed data requests.

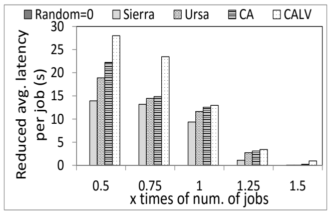


(a) Random data request distribution

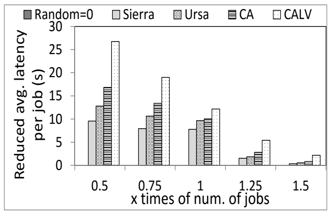


(b) Skewed data request distribution

Fig. 9: Performance on task latency with Delay scheduler.



(a) Random data request distribution



(b) Skewed data request distribution

Fig. 10: Performance on job latency with Delay scheduler.

require different amounts of computing resources. Therefore, some servers may still become overloaded and need to transmit data blocks to other servers. *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. 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. Also, without long-term load balance, load balancing for tasks' data servers when they become overloaded may not quickly offload their excess load, which still makes the tasks be allocated to other servers and requires data transfers. 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 using a truncated power-law distribution with the lower bound as 1 and the shape as 2, respectively. Figures 8(a) and 8(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 7(a) and 7(b). Figures 8(a) and 8(b) indicate that *CALV* achieves higher data locality performance and hence saves much more network load than other methods with skewed requests.

4.1.2 Performance of Task and Job Latency with the Delay Scheduler

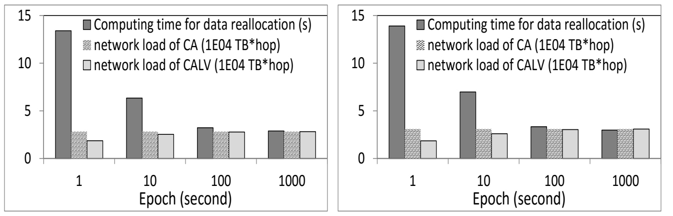
We measured the task latency and job latency that are the time elapsed from their submission to the end of their execution. Figures 9(a) and 9(b) show the reduced average latency per task of all methods compared to *Random* using the *Delay* scheduler. They show that the result follows 0=*Random*<*Sierra*<*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 7 and 8. The figures indicate that *CALV* generates the shortest task latency among all the methods. Figure 10 shows the reduced average job latency per job of all methods. The result follows the same order as Figure 9 due to the same reasons and shows that *CALV* generates the shortest job latency among all the methods.Compared to previous load balancing methods, *CALV* is novel in that it achieves long-term computing load balance. Thus, *CALV* reduces the probability that a task's assigned data server is overloaded and hence the probability of task re-assignment. As a result, *CALV* reduces network load for task re-assignment and task latency (without task-reassignment in the *Delay* scheduler).

4.1.3 Effect of the Time Length of Epochs

In this section, we test the effect of the time length of epochs on the effectiveness and overload of load balancing. We set the number of jobs as 1.5 times of that in the trace. Figure 11 shows the network load of *CALV* and *CA*, and the computation time for data reallocation of *CALV* with different epoch time lengths using the *FIFO* and *Fair* schedulers, respectively. *CALV* checks whether a server is overloaded after each epoch time based on the average workload per second during the epoch time. Both figures show that when the epoch length is 1 second and 10 seconds, the network load of *CALV* is lower than *CA* due to the same reasons as in Figure 7. When the epoch length increases to 100 and 1000 seconds, the network load of *CALV* increases and approaches that of *CA*, while the network load of *CA* maintains nearly constant. In *CALV*, when the epoch time is short, the load balancer can detect the server overload more accurately, which generates lower network load. As the epoch time increases, the accuracy that the load balancer detects server overload decreases. *CA* always aims to balance the computing workload per second during the load balancing period, so its network load performance is

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713



(a) Effect of epochs with FIFO (b) Effect of epochs with Fair

Fig. 11: Effect of the time length of epochs.

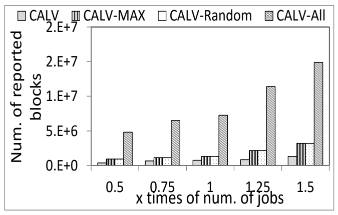


Fig. 12: Performance on reducing network overhead.

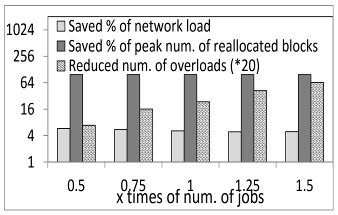


Fig. 13: Effectiveness of lazy data block transmission.

not affected by the epoch length. Figure 11(a) and 11(b) also show that the computation time decreases as the epoch time increases in *CALV*. With a longer epoch time, *CALV* has fewer epochs to consider in a time period, thus reducing computation time. The experimental results show that there is a trade off between the network load and computation time determined by the epoch time length in *CALV*.

4.2 Cost-Efficiency of Load Balancing

CALV also has a coefficient-based data reallocation method to choose as few data blocks 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 12 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 a part 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. 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 3.3 compared to *CALV-Max*, the computing capacities in source servers can be more fully utilized after data reallocation in *CALV* than in *CALV-Max*. The coefficient-based data reallocation method reduces the number of blocks reported to the load balancer. It also reduces the workload on the load balancer for load balancing.

4.3 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 13 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 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 13.

Figure 13 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, this 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.

4.4 System Overhead

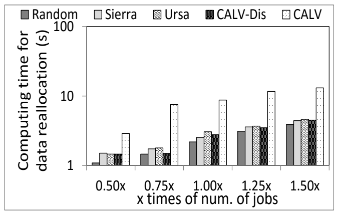


Fig. 14: System overhead.

In this section, we measured the computing time of different data allocation methods in order to show their system overhead in computing resource consumption. We also measured the computing time used by both servers and load balancer in *CALV* with the proximity-aware tree based distributed method, denoted by *CALV-Dis*. Figure 14 shows the computing time of all methods versus the number of jobs. It shows that *CALV* generates more computing time than other methods. That is because *CALV* additionally needs to find the priorities of data blocks to be reallocated and their destination servers by computing and comparing the overload contributions. *Random* generates the shortest computing time, because it assigns data without load balancing awareness. We can also see that *CALV-Dis* generates similar computing time with other methods due to its distributed and parallel process, which expedites the load balancing process. The figure indicates that *CALV* generates a slightly more computing time (e.g., 13.4s) for load balancing with all jobs in the trace [18]. However, with the proximity-aware tree based distributed load balancing

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

method, *CALV-Dis* generates similar computing time as other methods.

4.5 Performance of Task Reassignment Algorithm

In this section, we present the performance of the task reassignment algorithm, denoted by *CALV+TR*, compared with *CALV*. Figures 15(a) and 15(b) show the network load for data transfers in task running versus the number of jobs with the *FIFO* and *Fair* schedulers, respectively. They show that *CALV+TR* can save more network load than *CALV*. *CALV* only migrates data blocks to release the excess load of the overloaded servers. Some servers may not have available capacity to host more tasks, so that data transfers are needed when some tasks cannot be assigned to their data servers. *CALV+TR* takes advantage of data replicas in underloaded servers to release the excess load of overloaded servers. As a result, *CALV+TR* generates fewer data transfers and hence less network load in task running.

Figures 16(a) and 16(b) show the reduced average latency per task of all methods compared with *Random* versus the number of jobs with the *FIFO* and *Fair* schedulers, respectively. They show that *CALV+TR* reduces more task latency per task than *CALV*. As mentioned above, *CALV+TR* can achieve better load balanced state than *CALV*. With more overloaded servers, tasks have lower probability to be assigned to their data servers, which leads to longer average task latency. These figures indicate that *CALV+TR* has better performance than *CALV* in achieving load balance.

Figures 17(a) and 17(b) present the number of transferred blocks between servers versus the number of jobs with *FIFO* and *Fair*, respectively. These two figures show that *CALV+TR* generates fewer transferred blocks than that of *CALV*. *CALV+TR* releases the excess loads of overloaded servers by reassigning their tasks to other data servers of the tasks without data block transfers before it executes the *CALV* algorithm. Therefore, *CALV+TR* reduces the number of transferred blocks of *CALV*.

Figures 18(a) and 18(b) show the computing time of *CALV+TR* and *CALV* versus the number of jobs with the *FIFO* and *Fair* schedulers, respectively. They show that *CALV+TR* generates more computing time than that of *CALV*. This is because *CALV+TR* conducts the *TR* algorithm to reassign tasks from overloaded servers to their other data servers before conducts *CALV*. By reassigning a task to another data server of the task without the need of data transfer, the task reassignment algorithm is effective in improving *CALV* in terms of task latency, the number of transferred blocks and the computing time of the load balancing method.

4.6 Performance of Dynamic Load Balancing

In this section, we present the performance of the dynamic load balancing method (*DLB*). To generate the unexpected tasks, we randomly selected one task from the trace and added it into the job schedule after every 1000 regular tasks were completed. Since the performance of *DLB* does not depend on the types of job schedulers, we tested the performance of *CALV+DLB*, *CALV+TR+DLB*, *CALV+TR* and *CALV* with only the *FIFO* job scheduler. Figure 19(a) shows the network load for data transfer during job running versus the number of jobs. It shows that *CALV+DLB* has lower network load than that of *CALV*. Since *CALV+DLB* can redirect tasks to other data servers of the tasks or copy data blocks to underloaded servers. It can achieve load balanced state during running time. Due to the unexpected tasks, several servers may become overloaded in *CALV*. Therefore, more tasks need data transfers in running time in *CALV*, leading to higher network load. We see that the network load is approximately the same in *CALV+DLB* and *CALV+TR+DLB*. It means that *DLB* can effectively handle both the sever overloads caused by unexpected tasks and unsolved server overloads in offline load balancing. However, we need the offline *TR* algorithm because it can avoid data block movement between servers in load balancing. Figure 19(b) shows the reduced average latency per task compared with *Random* versus the number of jobs. It shows that *CALV+DLB* and *CALV+TR+DLB* have the best performance in task latency reduction. The reason is that in *DLB*, an overloaded server can redirect its tasks to other servers to become underloaded.

Figure 19(c) shows that the number of transferred blocks versus the number of jobs. It indicates that *CALV+DLB* has a slightly higher number of transferred blocks than *CALV*. This is because *DLB* is an additional step after *CALV*. In *DLB*, a server may need to replicate a data block to another server in order to redirect a task to the server, which generates block transfers. *CALV+TR* and *CALV+TR+DLB* generate much fewer block transfers. With the *TR* algorithm executed prior to *CALV* that reassigns tasks to their other data servers without moving data blocks, *CALV+TR* and *CALV+TR+DLB* greatly reduce the number of transferred blocks in *CALV+DLB*.

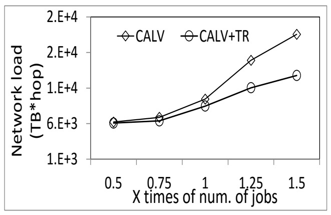
Figure 19(d) presents the computing time versus the number of jobs. We added the total computing time on the load balancer and the average computing time on each server as the computing time of *DLB*. The figure shows that *CALV+DLB* only slightly increases the computing time of *CALV*. *CALV+DLB* needs to additionally execute *DLB*. In *DLB*, each overloaded server tries to redirect tasks to their other data servers in a distributed manner to become non-overloaded, and if it fails to release excess load, it requests the load balancer to find underloaded servers to redirect tasks. *CALV+TR+DLB* generates slightly higher computing time than *CALV+TR* due to the same reasons. Since *TR* generates high computing time as explained previously, *CALV+TR+DLB* also generates high computing time though *DLB* only increases computing time slightly on each server. The dynamic load balancing method is effective in handling unexpected tasks.

4.7 Performance of the Proximity-aware Tree Based Distributed Load Balancing Method

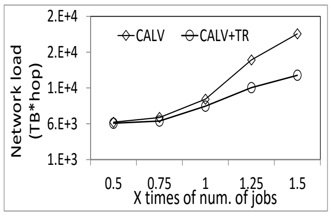
We then measured the effectiveness of the proximity-aware tree based distributed load balancing method. Figure 20 shows the saved percentage of the number of data blocks reported to the load balancer, the network overhead measured by $TB \cdot hop$ with the proximity-aware tree based method, and the saved percentage of network overhead calculated by $(no - no')/no$, where no and no' represent the network overhead generated by *CALV*

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

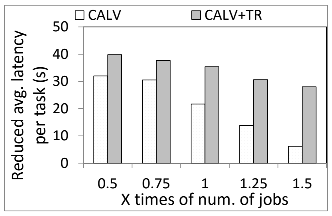


(a) Network load with FIFO

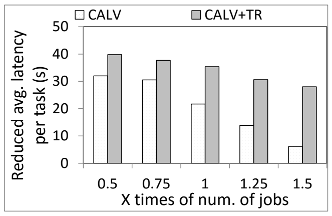


(b) Network load with Fair

Fig. 15: Data locality performance of CALV and CALV+TR.

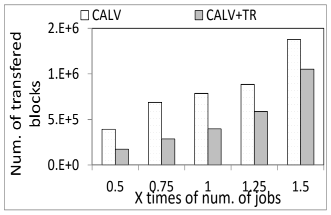


(a) Latency reduction with FIFO

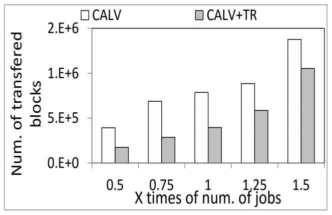


(b) Latency reduction with Fair

Fig. 16: Task latency reduction of CALV and CALV+TR.

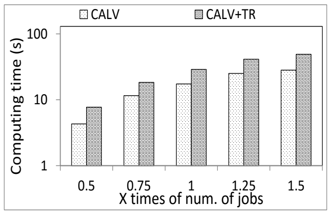


(a) Num. of transferred blocks with FIFO

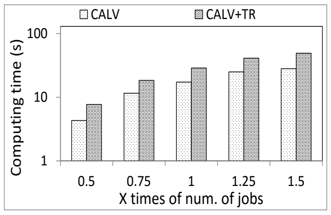


(b) Num. of transferred blocks with Fair

Fig. 17: The num. of transferred blocks of CALV and CALV+TR.

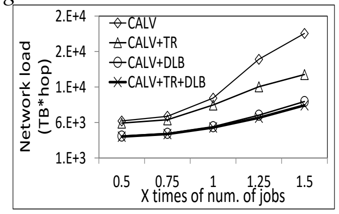


(a) Computing time with FIFO

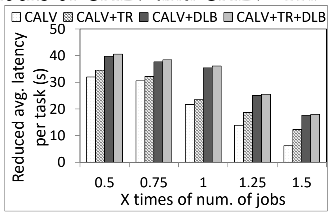


(b) Computing time with Fair

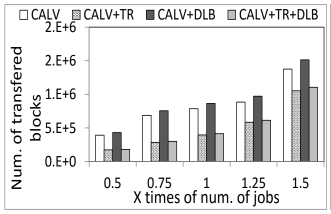
Fig. 18: Computing time of CALV and CALV+TR.



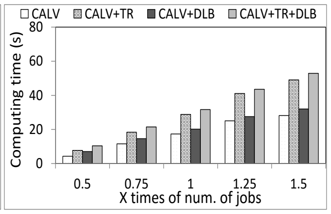
(a) Data locality performance



(b) Task latency reduction



(c) Num. of transferred blocks



(d) Computing time

Fig. 19: Performance of CALV+DLB and CALV+TR+DLB.

without and with this method, respectively. It shows that the network overhead is no larger than $20TB \cdot hop$ in the data reallocation. It also shows that this method can save at least 10% of network overhead in data reallocation, because it reallocates data with proximity-awareness. The figure also demonstrates that the distributed load balancing method can reduce at least 82% of the number of blocks reported to the load balancer because most of the overloads are resolved by the root servers in the sub-trees. The proximity-aware tree based distributed load balancing saves the network overhead in data reallocation and improves the scalability of the load balancer.

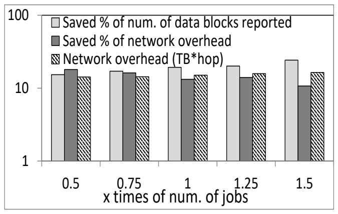


Fig. 20: Effect of proximity-aware tree based distributed load balancing.

5 Performance Evaluation on a Real Cluster

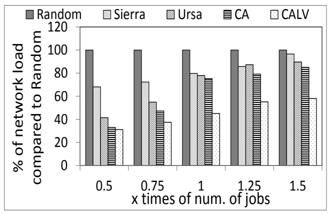
In this section, we present the experimental results on a real cluster. We implemented *CALV* and its comparison methods on the Apache Hadoop (version 1.2.1) on Palmetto [32], which is a computing cluster consisting of 771 8-core nodes. Due to the limitation of usage, we randomly selected 100 nodes to 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. The default size of a data block was set to 128MB. All other settings remain the same in the simulation. We measured the performance of *CALV* with the *FIFO*, *Fair* and *Delay* schedulers, respectively, by repeating the experiments in Sections 4.1.1 and 4.1.2.

Figures 21(a) and 21(b) show the percentage of the network load of all methods compared to *Random* 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 7(a) and 7(b), respectively. The results confirm that *CALV* can save more network load than all other methods with its computing load aware load balancing.

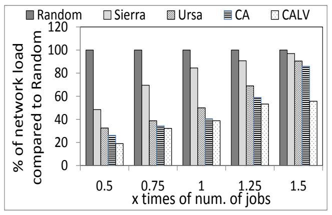
Figure 22 shows the reduced task latency of all methods compared to *Random* using the *Delay* scheduler. It demonstrates the same order and trend of all methods as in Figure 9(a) due to the same reason. It confirms that *CALV* can reduce the task latency and improve the task throughput with the *Delay* scheduler on a real computing cluster. Figure 23 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 12 due to the same reason. 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. The real cluster experimental results are consistent with the simulation results and confirm the higher performance of *CALV* compared with previous load balancing methods.

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713



(a) Network load with FIFO



(b) Network load with Fair

Fig. 21: Data locality performance on a Hadoop cluster.

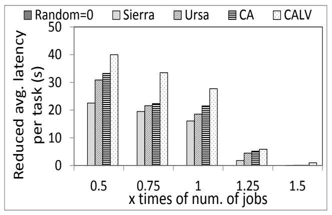


Fig. 22: Reduced task latency on a Hadoop cluster with Delay scheduler.

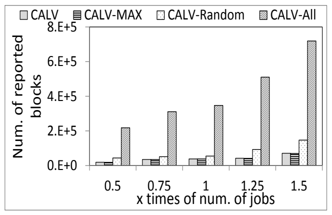


Fig. 23: Num. of reported blocks on a Hadoop cluster.

6 Related Work

Many research efforts have been devoted to data allocation in large-scale computing clusters. We classify these works into three groups for discussion below.

Random data allocation. The works in [2, 6, 38] 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, 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. Therefore, these methods cannot avoid 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. Ursa also works on power management and balancing, which is an important issue. In CALV, we reduce the network energy usage by improving the data locality. But to reduce or balance server computing energy usage is not our focus due to the existing large overhead. To cooperate any power management and balancing is in our future work. 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], 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. Fan *et al.* [14] found that more replicas give the job tracker more flexibility to assign tasks to nodes with local data copies, which leads to better load balance. Scarlett [15] and DARE [16] replicate data according to its popularity, so that popular data has more replicas in order to achieve load balance. However, all methods above only consider the data I/O workload, without considering the computing workload.

Wei *et al.* [17] considered the data servers' overload probability and their capacities (storage and computing) to calculate the minimum number of replicas of their data and determine data reallocation in order to maintain the required data availability to achieve load balance. However, it does not achieve long term load balance for computing workload of data blocks. *CALV* is the first work that balance the computing workload for the long term with the consideration of the differences of computing workloads associated with data blocks in servers.

7 Conclusion

Through an analysis of a Facebook cluster's job running trace, we show the importance of considering the computing workloads in load balancing for the long term. 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 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. *CALV* is further enhanced by a task reassignment algorithm and a dynamic load balancing method. Moreover, *CALV* depends on a proximity-aware tree based distributed load balancing method to improve its scalability and cost-efficiency. 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 further 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, IBM Faculty

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2016.2632713

Award 5501145 and Microsoft Research Faculty Fellowship 8300751. An early version of this work was presented in the Proc. of IEEE Bigdata 2015 [39].

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] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. Diskreduce: Raid for data-intensive scalable computing. In *Proc. of the 4th Annual Workshop on Petascale Data Storage*, 2009.

[15] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. G. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *Proc. of EuroSys*, 2011.

[16] C. Abad, Y. Lu, and R. Campbell. Dare: Adaptive data replication for efficient cluster scheduling. In *Proc. of ICCC*, 2011.

[17] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng. Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster. In *Proc. of CLUSTER*, 2010.

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

[19] Z. Li and H. Shen. Designing a hybrid scale-up/out hadoop architecture based on performance measurements for high application performance. In *Proc. of ICPP*, 2015.

[20] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. of VLDB*, 2012.

[21] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *Proc. of ICDCS*, 2010.

[22] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proc. of ECCS*, 2012.

[23] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proc. of SOCC*, 2012.

[24] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proc. of ICAC*, 2011.

[25] 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.

[26] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proc. of SIGCOMM*, 2015.

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

[28] 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.

[29] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou. Workload characterization on a production Hadoop cluster: A case study on Taobao. In *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*, 2012.

[30] 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.

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

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

[33] 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].

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

[35] 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.

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

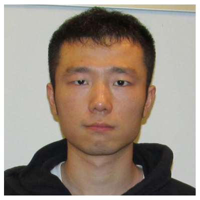
[37] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs Scale-out for Hadoop: Time to rethink? In *Proc. of SoCC*, 2013.

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

[39] G. Liu, H. Shen, and H. Wang. Computing Load Aware and Long-View Load Balancing for Cluster Storage Systems. In *Proc. of IEEE BigData*, 2015.

Guoxin Liu received the BS degree in BeiHang University 2006, and the MS degree in Institute of Software, Chinese Academy of Sciences 2009. He is currently a Ph.D. student in the Department of Electrical and Computer Engineering of Clemson University. His research interests include distributed networks, with an emphasis on Peer-to-Peer, data center and online social networks.

Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University, China in 2000, and the MS and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Associate Professor in the ECE Department at Clemson University. Her research interests include distributed computer systems and computer networks with an emphasis on P2P and content delivery networks, mobile computing, wireless sensor networks, and cloud computing. She is a Microsoft Faculty Fellow of 2010, a senior member of the IEEE and a member of the ACM.



Haoyu Wang received the BS degree in University of Science & Technology of China, and the MS degree in Columbia University in the city of New York. He is currently a Ph.D student in the Department of Electrical and Computer Engineering of Clemson University. His research interests include data center, cloud and distributed networks.

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.