Log-assisted Straggler-aware I/O Scheduler for High-End Computing

Neda Tavakoli
Dept. of Computer Science
Texas Tech University
Lubbock, TX, USA 79409
Email: neda.tavakoli@ttu.edu

Dong Dai
Dept. of Computer Science
Texas Tech University
Lubbock, TX, USA 79409
Email: dong.dai@ttu.edu

Yong Chen
Dept. of Computer Science
Texas Tech University
Lubbock, TX, USA 79409
Email: yong.chen@ttu.edu

***Abstract*—Object-based parallel file systems have emerged as promising storage solutions for high-end computing (HEC) systems. Despite the fact that object storage provides a flexible interface, scheduling highly concurrent I/O requests that access a large number of objects still remains as a challenging problem, especially in the case when stragglers (storage servers that are significantly slower than others) exist in the system. An efficient I/O scheduler needs to avoid possible stragglers to achieve low latency and high throughput. In this paper, we introduce a log-assisted straggler-aware I/O scheduling to mitigate the impact of storage server stragglers. The contribution of this study is threefold. First, we introduce a client-side, log-assisted, straggler-aware I/O scheduler architecture to tackle the storage straggler issue in HEC systems. Second, we present two scheduling algorithms that can make efficient decision on scheduling I/Os while avoiding stragglers based on such an architecture. Third, we evaluate the proposed I/O scheduler using simulations. The simulation results have confirmed the promise of the newly introduced log-assisted straggler-aware I/O scheduler in large-scale HEC systems.**

I. Introduction

The shift toward data-driven scientific discovery and innovation has made many high-end computing (HEC) applications more highly data intensive than ever before. The I/O performance is considered an increasingly critical factor that determines the overall HEC system performance. In the meantime, object-based parallel file systems [1], in which files are represented as a set of objects stored on object-based storage devices (OSDs) [2], [3], managed by object storage servers (OSSs), have been increasingly deployed on large-scale high-end/high-performance computing systems due to their merits of improved scalability, manageability, and performance [4], [5], especially when highly concurrent I/O requests occur.

Developing a highly efficient I/O scheduler in the object-based storage systems is arguably critical and a well-acknowledged challenge. A considerable amount of work has been done in this space [2], [6]–[11]. Among them, the *straggler* problem [12]–[16], which occurs when some of OSSs take a much longer time in responding to I/O requests than other servers, has drawn lots of attentions recently.

The occurrence of stragglers has significant affects on I/O performance of object storage systems. Since in HPC applications, clients normally need to synchronize after each I/O phase [3], [17], the overall I/O performance will be determined by the longest one, which in turn is determined by the *slowest* object storage server. In general, the slow storage servers (i.e., stragglers) can be divided into two categories: long-term stragglers and short-term stragglers. Long-term stragglers can be slow for hours, days and even in the worst-case forever. The main reasons of occurring this type of straggler can be outdated hardware, hardware failures and even software bugs. On the other hand, short-term stragglers last only for minutes or less. The main reasons of this type of straggler are interference or resource contention between applications.

Fig. 1 illustrates an example of how a straggler in object storage servers can affect the I/O performance. Assuming application processes issue three I/O requests, and each line on the top of the figure represents part of each I/O to access OSSs. The bottom part of the figure illustrates three OSSs in this example, assuming that OSS-0, in shaded pattern, is a straggler. If any part of an I/O hits the straggler server OSS-0, the entire I/O suffers from the straggler problem because the I/O cannot be completed until the slowest server OSS-0 finish servicing its part of I/O access. The existence of storage server stragglers can significantly reduce the productivity of the HEC system. Even though an asynchronous I/O can decouple the computation and I/O and is helpful in some cases, but it is not always possible. For instance, if the computation depends on the I/O reads, or the simulation output is so large that these data have to be flushed to storage systems before the computation can carry on, the application processes have to be blocked utill the I/O is completed. The storage server straggler problem can be catastrophic in projected extreme-scale systems, as the large-scale storage system significantly increases the possibility of the existence of a straggler in storage servers.

In order to mitigate the impact of storage server stragglers, a straggler-aware I/O scheduling mechanism is imperatively desired for increasingly popular object-based parallel file systems. Numerous recent studies have focused on this problem, including our prior research of a two-choice randomized I/O scheduling strategy [18]. However, these strategies suffer from expensive probing messages and communication overhead, as they need to probe storage servers' status and outstanding workloads to detect the existence of stragglers and adopt a randomized scheduling strategy to avoid the straggler. In

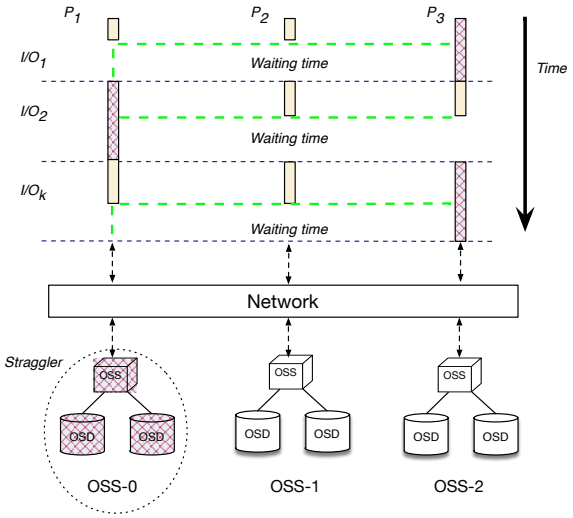

Fig. 1: An illustration of how a straggler in object storage servers can affect the I/O performance in HEC.

this research study, to overcome the limitation of probing message based straggler-aware I/O scheduling, we introduce a *log-assisted straggler-aware I/O scheduling*. The fundamental idea is that, the I/O scheduler at the client side maintains a log of I/O requests, servers' status, and past scheduling decisions, so that the I/O scheduler client can make an optimized scheduling decision when stragglers exist, without incurring expensive probing messages and communication cost. Based on this new idea, we also introduce two different scheduling policies that leverage this log information to make the scheduling decision. To validate the newly introduced log-assisted straggler-aware I/O scheduling strategy and to verify the effectiveness of different scheduling policies, we have conducted simulation evaluation, and these simulation results confirm that the newly proposed scheduler is effective in improving I/O performance with avoiding storage server stragglers and significantly reducing probing messages and communication cost. The contribution of this research study is three-fold:

- Introduce a log-assisted straggler-aware I/O scheduling for object-based parallel file systems with the log design and the mechanism of maintaining the logs at the scheduler client;
- Introduce two scheduling algorithms based on the log-assisted straggler-aware scheduling mechanism that can be used in parallel file systems;
- Conduct simulation evaluations to validate the concept and analyze the effectiveness of a log-assisted straggler-aware scheduler and two different scheduling algorithms.

The rest of this paper is organized as follows. Section II gives a detailed description of the proposed client-side log-assisted I/O scheduler and proposed scheduling policies. Section III reports the experimental results. Section IV discusses relevant work and compares with this study. We conclude this research and describe possible future work in Section V.

II. Architecture and Design

A. System Architecture

The overall architecture of the proposed log-assisted straggler-aware I/O scheduler is illustrated in Fig. 2. The proposed scheduler runs on the client-side (compute nodes), as an I/O scheduler serving each I/O request. A key component of the proposed scheduler is a *client-side server statistic log*, which will be described in detail in the next subsection. In the object storage server side, we reuse the components designed and implemented in our previous work [18]. Specifically, we have a *redirect table* and *metadata maintainer thread* running on the object storage servers. Among them, the redirect table is used to remember the default location of data objects. This is necessary since we will dynamically place I/O requests to different object storage servers based on the scheduling decision. Obtaining a distributed redirect table in each object storage server is more scalable compared with storing every redirection into metadata service of the parallel file systems. Another component, the metadata maintainer thread runs in the background to move the redirected objects back to their default location when the file systems are idle. This helps keep the data location consistent with metadata in parallel file systems and hence improves the performance of further reads.

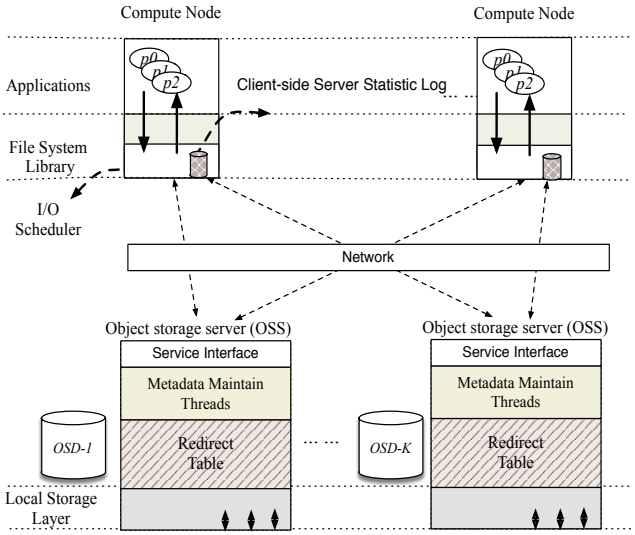

Fig. 2: Overall system architecture of a log-assisted straggler-aware I/O scheduler.

Fig. 3 further illustrates an example of how the redirect table and metadata maintainer thread work. It shows a data fragment (in black), which should be placed on OSD_0 but is scheduled to be written to OSD_2 as OSD_0 is found to be a straggler. After the scheduler arranges the data fragment to be written to OSD_2, a new entry will be created in the redirect table of OSD_0 indicating that the current location of the segment is on OSD_2. The dashed red line indicates the metadata maintainer thread periodically runs to retrieve this

segment back and deletes the entry in the redirect table. More details can be found in our previous work [18].

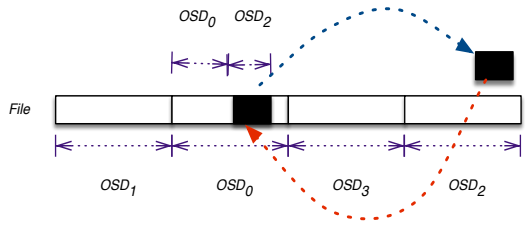

Fig. 3: Redirect table for straggler-aware I/O scheduler.

B. Scheduling Model

In the proposed architecture, for each application run with multiple processes, the I/O requests from each process are combined and scheduled to access object storage servers together in a fashion similar to the collective I/O. These I/O requests are not necessarily from the same process and can be from a group of processes, similar as in collective I/O. Specifically, the concurrent I/O requests issued in one client are queued temporarily and served in the group. We define these multiple I/O requests served together as a *time series*, similar as in [19].

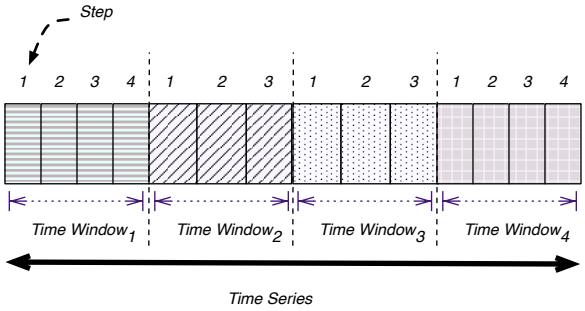

Fig. 4: Time series, time window, and step number in scheduling.

We show the concept of time series in Fig. 4. Time series is divided into consecutive segments called *time window*, which contains a fixed time interval. Each time, the proposed I/O scheduler will serve all buffered requests before moving to the next time window. Inside one time window, a number of I/O requests can be queued. We divide queued I/O requests in each time window into multiple steps. Each step will contain the I/O requests on the same object to reduce unnecessary network communications.

C. Client-side Server Statistic Log

A client-side server statistic log is introduced as a core component of a new log-assisted straggler-aware I/O scheduler. As shown in Fig. 2, it works inside the I/O scheduler on the client-side.

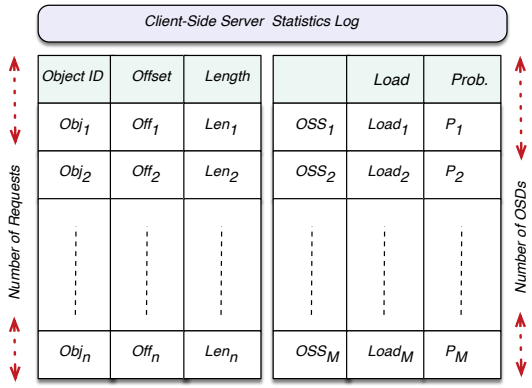

Fig. 5: Client-side server statistic log.

1) Data Structure of Client-Side Server Statistic Log: We show the detailed structure of the client-side server statistic log in Fig. 5. It consists of two tables: an *I/O request table*, shown as the left part of Fig. 5, and a *server statistic table*, shown in the right part. In the I/O request table, each request forms a row, which consists of three fields: 1) *Object ID*, which is a unique identifier (a very large index number) of each object; 2) *Offset*, which denotes the starting offset of the I/O request from the beginning of the object; and 3) *Length*, which is the size of the I/O request (in bytes). On the other hand, the server statistic table consists of statistics of each object storage server. It is generated based on scheduling decisions from previous time windows. The *Load* field is an expectation of the amount of I/O requests (in terms of bytes) in each object storage server. Different from our previous work [18], in this research, we do not probe the servers to retrieve their realtime loads; instead, we use this server statistic table to update and record current load of each storage server based on previous scheduling decisions. We use such load to calculate the probability of selecting a corresponding storage server for a given I/O request, which is maintained and updated in the *Prob* field. In the next subsection, we introduce how to update the server statistic table and also calculate the probability.

2) Maintaining Servers Statistic Table: First, after serving each I/O request in a single step, the server loads in the server statistic table will be updated based on the following formula:

$$l = l' + Len \tag{1}$$

where l is the expected server load at the end of each step, l' represents the server load from the previous step, and Len is the length of scheduled I/O requests. At the beginning of the next step, we will calculate the probability of selecting server i and use such probability to choose the best storage server to schedule the I/O request. The probability is calculated according to the following formula:

$$p_i = p'_i * e^{-l_i} \tag{2}$$

where p'_i indicates the current probability of selecting server i, and l equals to the updated load of server i. The initial

p'_i is calculated based on the default round-robin scheduling strategy: given there are M object storage servers, each server is equally assigned a $p'_i = \frac{1}{M}$. Through calculating such a probability for each object storage server, we are able to choose one as target (the detailed selection algorithms are introduced in the next subsection). After making a scheduling decision, we will need to update the server statistic table based on Equation 1. In addition to updating the load of the selected server, we also need to update the probability of choosing another server to prepare for the next scheduling. In fact, the summation of probabilities of choosing all storage servers should equal to 1. This also requires us to update their probabilities too. The following equation shows how this is done:

$$p_j = p_j^{'} + \frac{(p'_i - p'_i * e^{-l_i})}{M-1}, j \neq i \quad (3)$$

where M indicates the total number of storage servers, i is the server that is chosen to serve the current I/O request, and j refers to all other object storage servers. This equation indicates that we evenly re-distribute the decreased probability of server i to all other object storage servers (j). We use this strategy to maintain the probability. Note that, in Equation 2, we use exponential distribution to calculate the probability of a server being chosen according to their current loads. We use the exponential distribution [20] to calculate it because this distribution describes the behavior of balanced scheduling. Specifically, it indicates that servers with high loads (which could be very high due to imbalanced requests) should be considered with lower probability to serve incoming I/O requests to avoid the straggler. On the other hand, the server with lighter loads (which could approach to 0) should be considered with a much higher probability of being chosen. If a server does not contain any load (i.e., 0), it should be selected with the highest priority. Other distributions can possibly describe similar behaviors; however, since that is not our focus in this research, we leave the investigation of distributions as a future work.

D. Scheduling Algorithms

Based on the information stored in the server statistic log, we propose two new scheduling algorithms to leverage this statistic log to develop an efficient straggler-aware I/O scheduler. Before introducing the proposed scheduling algorithms, we first introduce the base-line algorithm, a round-robin (RR) algorithm. This strategy is widely used in modern parallel file systems like PVFS [21], GPFS [22], and Lustre [23]. For an I/O request on object i, RR schedules it to the storage server $i \mod M$, where M indicates the total number of object storage servers to be fair among multiple I/O requests and servers, as well as to avoid starvation. It is not trivial to see that the RR strategy does not work well with stragglers as it does not consider current performance of storage servers. Hence, it may still choose the overloaded servers even there are lightly-loaded servers that can be chosen.

We introduce two straggler-aware scheduling algorithms based on the proposed client-side server statistic log. The first one, called *Max Length - Min Load (MLML)* scheduling algorithm, directly uses logged server statistics to direct scheduling; i.e., it will select the server with lighter workloads with a higher probability. The second scheduling algorithm, called *Two Random from Top Half (TRH)*, takes advantage of the random strategy in addition to server statistic log. Our evaluations confirm that these two scheduling algorithms can both reduce the impact of existing stragglers and avoid generating new stragglers. More details of their comparisons can be found in the evaluation section. We discuss each of these two algorithms in detail below.

1) Max Length - Min Load (MLML) scheduling algorithm: This scheduling strategy uses the statistic server log to sort the storage servers based on their probabilities from the highest to the lowest. At the same time, the I/O requests are sorted from the maximum length to the minimum one. These two sorted lists are processed sequentially, from the top to the bottom of lists then starting again from the top of the lists in a circular manner. As a result, the maximum length I/O request will be scheduled to the server with lighter load and the minimum length I/O request will be distributed to the server with heavier load.

Algorithm 1 Max Length-Min Load algorithm (MLML)

```
1:  procedure MLML(SERVERS,REQUESTS,THRESHOLD)
2:      sortedServers = sort(servers);
3:      sortedRequests = sort(requests);
4:      index=0;
5:      while need_schedule() do
6:          default_oss = requests[index] mod M;
7:          target_oss = sortedRequests[index] mod M;
8:          benefit = load(default_oss) - load(target_oss);
9:          if benefit ≤ threshold then
10:             return target_oss;
11:         else
12:             return default_oss;
13:         index++;
```

We describe the pseudocode of the MLML scheduling strategy in Algorithm 1. In this algorithm, two sorted lists *sortedServers* and *sortedRequests* store a sorted list of servers and I/O requests, respectively. The default_oss describes the default location of requested objects. As we have described, such a server is selected through the default RR strategy. The target_oss, on the other hand, represents the storage server calculated based on this MLML scheduling algorithm. If the target_oss and the default_oss are not the same server, we then need to consider whether it is worth of scheduling this request to the target_oss as this might increase read overhead due to the redirection. We introduce a user-defined threshold to indicate how much overhead is acceptable. Specifically, we compare loads of those two servers and if the benefit of choosing the target_oss over the default_oss is larger than the threshold, the selection is acceptable; otherwise the default server will still be selected. The rationale behind this algorithm is that a server with lighter load has a lower chance of

being a straggler. By scheduling larger requests on the lighter server, we can reduce or minimize the impact of existing stragglers while the chance of generating new stragglers can be significantly decreased.

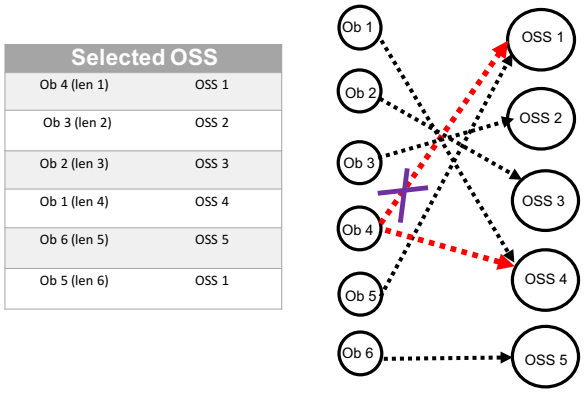

Fig. 6: Example of MLML algorithm.

Fig. 6 illustrates an example of how this MLML algorithm works. In this example, there are six objects and five object storage servers. As shown in the figure, I/O requests are sorted based on their length, and OSSs are sorted based on their probability of being stragglers from the highest to the lowest. Based on the MLML strategy, these I/O requests are assigned to OSSs based on the sorted list. In this example, all selected OSSs by the scheduling algorithm provide benefits greater than the threshold, except for the ob 4, which will be scheduled to the default server (i.e. distributed to OSS4 instead of OSS1). Note that the redirect table is used to keep track these writes and maintains metadata consistency.

2) Two Random from Top Half (TRH) algorithm: This algorithm considers randomized choices based on the server statistic log to further mitigate the impact of stragglers. The rationale of using randomized choices in this algorithm is that it provides more possible options for selecting servers and helps to spread the requests to be more balanced. In this scheduling algorithm, the sorted list of object storage servers is also used, same as in the MLML algorithm. Specifically, for a given I/O request, two object storage servers will be randomly chosen from the top half of all object storage servers. The top half indicates $\frac{M}{2}$ servers that have lighter loads. We also use the same threshold as in the previous algorithm to decide whether choosing the calculated target_oss or the default_oss. Algorithm 2 shows the pseudocode of the TRH algorithm.

Fig. 7 further illustrates an example showing how this algorithm works. In this example, for each object, two OSSs are randomly selected as potential targets and the one with the lighter load will be chosen according to the Algorithm 2. In this specific example, we show that all objects are scheduled to the lighter load server based on their two random choices except for ob 3 that is scheduled to its default_oss because its default_oss (i.e., OSS_3) has a similar load compared to OSS_5 (the algorithm considers it is not worth of scheduling it to another location).

Algorithm 2 Two Random from top Half (TRH) Algorithm

```
1:  procedure TRH(SERVERS,REQUESTS,THRESHOLD)
2:      sortedServers = sort(servers);
3:      sortedRequests = sort(requests);
4:      index=0;
5:      while need_schedule() do
6:          default_oss = requests[index]  mod  M;
7:          ross_1=random_oss();
8:          ross_2=random_oss();
9:          target_oss =find_min_load(ross_1, ross_2);
10:         benefit = load(default_oss) - load(target_oss);
11:         if benefit ≤ threshold then
12:             return target_oss;
13:         else
14:             return default_oss;
15:         index++;
```

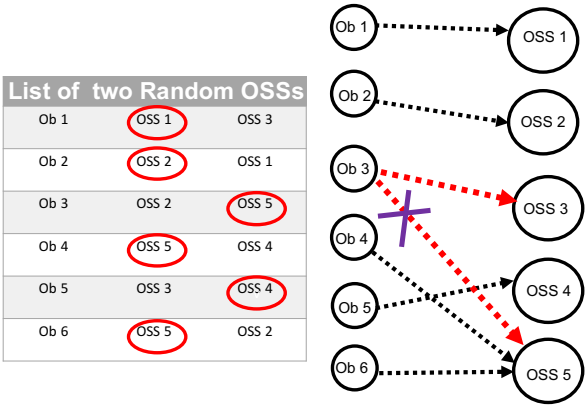

Fig. 7: Example of TRH algorithm.

III. Evaluation and Analysis

We have evaluated the proposed I/O scheduler and scheduling algorithms using simulation. In the simulation, we used synthetic workloads generated based on real-world traces. We constructed the synthetic workload based on combining three different types of I/O requests: large I/O (each request is greater than $O(10MB)$), medium I/O (each request is between 4MB and 10MB) and the small one (less than 4MB). These three categories of requests are chosen based on our observation of the typical scientific applications running on HEC systems. In addition, the initial I/O loads of all OSSs are generated based on a normal distribution with a small standard deviation, which simulates a roughly even workload distribution at the beginning. The simulation is considered as running in an HEC cluster with 100 object storage servers and 200 compute nodes. In all test cases, we run the simulation multiple times (100) and calculate the average load of each OSS under the synthetic workload with the proposed log-assisted straggler-aware scheduler and with the MLML and TRH scheduling algorithms, respectively. The total number of I/O requests simulated was 2,000, with various sizes for each

request; for instance, the total data written for all medium I/O case can be up to 20GB. The total data written for all large I/O case can be between $O(20GB)$ and $O(2TB)$ depending on the size of each request.

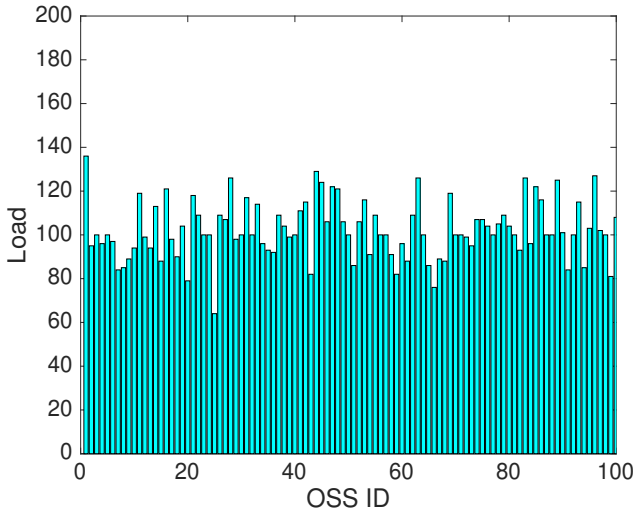

Fig. 8: Load distribution with the RR scheduling algorithm.

We consider the round-robin (RR) algorithm as the baseline case and compare it with the proposed straggler-aware scheduler with two algorithms. In Fig. 8, we first illustrate the load of each server after scheduling all I/O requests using the round-robin policy. The x-axis shows the storage servers, and the y-axis represents the load of each storage server (i.e, write data size in MB).

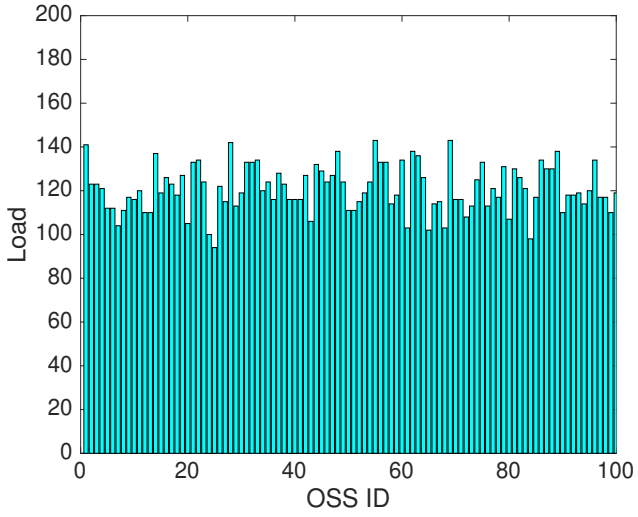

Fig. 9: Load distribution with the log-assisted straggler-aware scheduling with the MLML algorithm.

Fig. 9 plots the load distribution of storage servers with the log-assisted straggler-aware scheduler after scheduling all requests using the MLML algorithm. As the figure shows, since the MLML algorithm rotates the requests to servers based on their loads, this algorithm is able to achieve a better load balance across different storage servers compared with Fig. 8. We further show the result of load distribution with the TRH algorithm in Fig. 10. Since this algorithm randomly chooses two servers from the top half of the lighter loaded storage servers and selects the better one, it can effectively avoid stragglers to achieve better balance compared with the round-robin strategy. However, it has large variances than that of the MLML algorithm mostly due to the randomness.

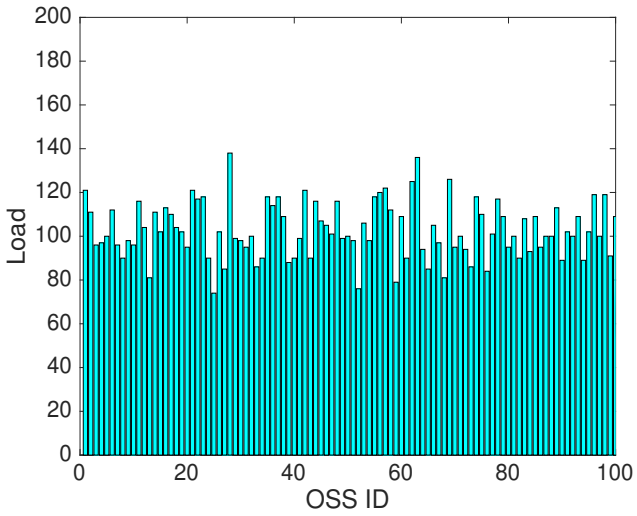

Fig. 10: Load distribution with the log-assisted straggler-aware scheduling with the TRH algorithm.

To further validate and verify the effectiveness of the proposed log-assisted straggler-aware I/O scheduler, in this series of evaluations, we manually inject a number of storage servers as stragglers. We expect the straggler-aware scheduler to be able to avoid hitting existing stragglers and also avoid generating new stragglers.

In these tests, we assigned 10% of the total object storage servers as stragglers by adding extra loads on those selected servers. Specifically, we injected 5 times more load compared with the average loads assigned on other storage servers.

We report the results in Fig. 11, Fig. 12, and Fig. 13. In these figures, the x-axis indicates all the possible loads of storage servers after scheduling; the y-axis shows the maximal IO requests scheduled onto storage servers that have the corresponding load as the x-axis shows. We only show the maximal I/O requests since there might be multiple servers having the same load after scheduling, and the one receiving the most I/O requests actually determine the overall performance as we have described.

We expect an effective straggler-aware scheduler should be capable of avoiding stragglers; in other word, the number of I/O requests falling into the slow storage servers (stragglers) should be close to zero. Based on the simulation results

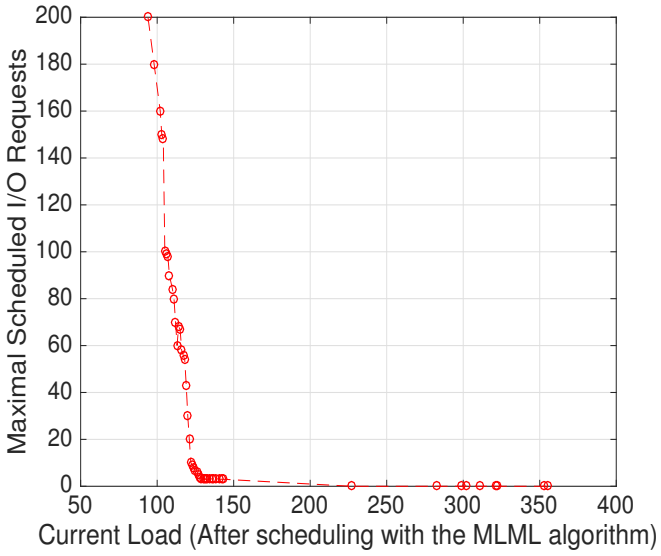

Fig. 11: Maximal scheduled I/O requests on servers with different loads with the MLML algorithm.

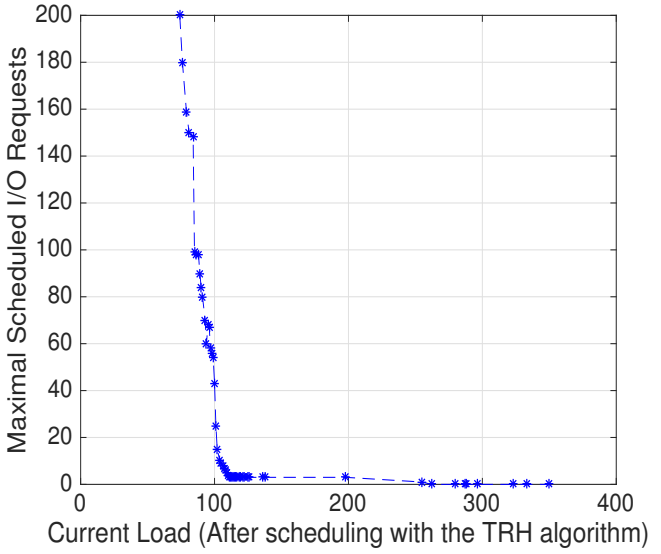

Fig. 12: Maximal scheduled I/O requests on servers with different loads with the TRH algorithm.

reported in Fig. 11 and Fig. 12, the log-assisted straggler-aware scheduler with MLML and TRH algorithms are capable of achieving that since they do not schedule any I/O request onto servers with load greater than 200 in these test cases. We also compare the proposed MLML and TRH scheduling algorithms with the base-case round-robin scheduling. Fig. 13 plots these results and shows this comparison. From this figure, we can observe that the RR strategy hits highly loaded servers (over 300). On the other hand, the log-assisted straggler-aware scheduler with both the MLML and TRH scheduling algorithms are able to avoid such stragglers. Between the MLML and TRH algorithm, we can notice that the TRH scheduling algorithm tends to place loads on lighter loaded servers (the curve is on the left side of the MLML curve). This behavior is due to the fact that randomized scheduling choices introduce more options for selection and better promote to spread the requests.

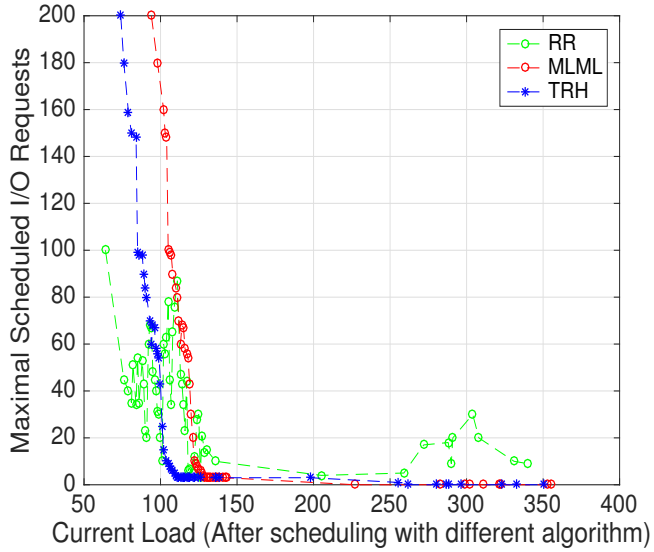

Fig. 13: Comparison of RR, MLML, and TRH scheduling algorithms.

These simulation results show that using the client-side server statistic log and the proposed scheduling algorithms can avoid storage server stragglers and improve the scheduled workload balance among storage servers.

IV. Related Work and Comparison

I/O scheduling has been extensively studied in earlier research efforts. The straggler problem has been well acknowledged as one of critical issues in I/O scheduling. Current research is heavily focused on trying to avoid stragglers, including our previous work of an I/O scheduler based on a two-choice randomized scheduling that dynamically places write operations to mitigate the straggler problem in high-end/high-performance computing systems [18]. This paper is different from two-choice approaches as we avoid the probing latency through using client-side logs. ADIOS [24] introduced an adaptive I/O method to reduce the I/O variability within a single application. It gathers all processes inside the application in a group and each group writes in a particular storage server. CALCioM [25] mitigates I/O interferences in HPC systems through a cross-application coordination. It allows different applications to communicate and coordinate their I/O strategies to avoid congestion. Our study is different from it as we are introducing a client-side scheduler that can transparently identify those patterns and coordinate to mitigate stragglers and hence alleviate the interferences. There are also

series of research focusing on identifying I/O access patterns of applications and using such information to mitigate stragglers and improve the performance [26]–[28]. Our research is different from them as we do not need to know the details about each application, instead, by checking the requests and previous served I/O logs, we are able to dynamically schedule the I/O requests to mitigate stragglers.

The two-choice algorithm is also used in Sparrow, a distributed task scheduler [29]. It uses multiple distributed schedulers on a cloud-based platform to schedule a large number of short tasks. Sparrow avoids the straggler problem by probing multiple nodes for each task and scheduling that task on the node with the least overhead. The Sparrow scheduler is specifically designed for cloud systems and workloads though.

Another straggler mitigation technique is introduced in Dolly with a proactive straggler mitigation approach [30]. It uses a proactive database provisioning scheme and leverages virtual machine cloning in cloud platforms. It accomplishes this by using the power-law distribution of job sizes to launch different clones of each task. Furthermore, it attempts to predict stragglers by waiting and observing the system behavior before scheduling a task.

Yet another straggler mitigation technique is presented in Mantri [16], which moves tasks from straggling resources and assigns them to less overloaded resources. It monitors tasks and attempts to avoid stragglers based on their causes. It accomplishes this goal in three primary steps. First, speculative copies of straggler tasks are executed while being aware of resource constraints and work imbalance. Second, all tasks are placed based on the locations of their data sources. Last, once a task is completed, its output (intermediate data) is replicated on other resources with lighter workloads.

One of the most widely used techniques for mitigating the impact of stragglers is a speculative execution technique. Speculative execution means that, for those tasks that are stragglers or are likely to be stragglers, extra copies of tasks are executed. Among these copies, the one that finishes first is chosen [16], [31], [32]. Many systems and studies use speculative executions to address the straggler problem [16], [31], [33], [34], including a cloning approach [33], [35] and a straggler-detection-based approach [36], [37]. The primary difference between these two approaches is how/when they launch their speculative task copies. In the cloning approach, the initial task is scheduled in parallel with extra copies, and the task that finishes first is used for the next scheduled task. In the straggler-detection based approach, the extra copies are only launched if stragglers are detected. One speculative task scheduler, LATE [34], implicitly assumes that cluster nodes are homogeneous and linearly make progress. It uses these assumptions to speculatively re-execute tasks that appear to be stragglers. In fact, it focuses on reducing the response time of scheduling by speculating which running task can be overtaken. Any task that may be overtaken will then be re-executed to avoid stragglers. Hopper [38], [39], is another example of a job scheduler that uses speculative execution to mitigate the straggler problem. The key idea of Hopper is that it must predict the speculation requirements of jobs and then dynamically launch extra copies on alternative machines. The downside of speculative execution is that, while it mitigates the impact of stragglers, it consumes valuable system resources due to redundant executions. Furthermore, it does not coordinate resources for concurrent jobs well.

In this research, we introduce a log-assisted straggler-aware I/O scheduling. It is different from these existing studies as it addresses the storage server straggler problems in object based storage systems. It is specifically designed for high-end computing systems where high concurrency is the norm and uses client-side logs to reduce probing messages that are required in two-choice or similar scheduling algorithms. It avoids redundant executions as required in the class of speculative techniques. To the best of our knowledge, this study is a first research effort that introduces a log-assisted straggler-aware scheduling approach.

V. Conclusion and Future Work

Many high-end/high-performance computing applications have become highly data intensive and the needs of highly efficient storage systems to better support scientific discovery substantially grow over years. In this research, motivated by the imperative needs of better dealing with storage server stragglers (servers that take much longer time in responding to I/O requests than other servers due to transient overloaded requests, imbalanced accesses, or even hardware/software transient failures), we introduce a new log-assisted straggler-aware I/O scheduler. We have presented the idea, design, and evaluation of such straggler-aware I/O scheduling strategy. We have introduced a client-side server statistic log to maintain I/O requests and servers status so that the I/O scheduler client can make an optimized scheduling decision when stragglers exist, without incurring expensive probing messages as in the existing straggler-aware I/O scheduling methods. We have presented two straggler-aware scheduling algorithms based on the proposed client-side server statistic log, a Max Length - Min Load (MLML) algorithm and a Two Random from Top Half (TRH) algorithm. Our evaluations confirm that these two scheduling algorithms can reduce the impact of existing stragglers and avoid generating new stragglers. The evaluations confirm that the log-assisted straggler-aware scheduling achieves a better load balance on storage servers while avoiding stragglers.

We plan to further investigate and develop a prototype of integrating the proposed I/O scheduling into existing parallel file systems. In addition, while the current simulation shows the solution improves the I/O performance, we believe that more optimizations can be made with the server statistic logs, and we will further investigate this aspect in future as well.

Acknowledgment

This research is supported in part by the National Science Foundation under grant CCF-1409946 and CNS-1338078. We greatly appreciate the time and efforts by the referees in reviewing this paper and the valuable suggestions offered.

REFERENCES

[1] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based Storage," *Communications Magazine, IEEE*, vol. 41, no. 8, pp. 84–90, 2003.

[2] Y. Liu, R. Figueiredo, D. Clavijo, Y. Xu, and M. Zhao, "Towards Simulation of Parallel File System Scheduling Algorithms With PFSsim," in *Proceedings of the 7th IEEE International Workshop on Storage Network Architectures and Parallel I/O*, 2011.

[3] J. M. Del Rosario, R. Bordawekar, and A. Choudhary, "Improved Parallel I/O Via a Two-phase Run-time Access Strategy," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 5, pp. 31–38, 1993.

[4] D. Dai, X. Li, C. Wang, M. Sun, and X. Zhou, "Sedna: A memory based key-value storage system for realtime processing in cloud." in *CLUSTER Workshops*, 2012, pp. 48–56.

[5] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object Storage: The Future Building Block for Storage Systems," in *Local to Global Data Interoperability-Challenges and Technologies, 2005*. IEEE, 2005, pp. 119–123.

[6] R. Jain, K. Somalwar, J. Werth, and J. C. Browne, "Heuristics for Scheduling I/O Operations," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 3, pp. 310–320, 1997.

[7] F. Chen and S. Majumdar, "Performance of Parallel i/o Scheduling Strategies on a Network of Workstations," in *Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference on*. IEEE, 2001, pp. 157–164.

[8] D. Durand, R. Jain, and D. Tseytlin, "Parallel I/O Scheduling Using Randomized, Distributed Edge Coloring Algorithms," *Journal of parallel and distributed computing*, vol. 63, no. 6, pp. 611–618, 2003.

[9] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante, "The Impact of I/O on Program Behavior and Parallel Scheduling," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1. ACM, 1998, pp. 56–65.

[10] Y. Wiseman and D. Feitelson, "Paired Gang Scheduling. Parallel and Distributed Systems," *IEEE Transactions on*, vol. 14, no. 6, 2003.

[11] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante, "Models of Rarallel Applications With Large Computation and I/O Requirements," *Software Engineering, IEEE Transactions on*, vol. 28, no. 3, pp. 286–307, 2002.

[12] F. Wang, S. A. Brandt, E. L. Miller, and D. D. Long, "Obfs: A file system for object-based storage devices." in *MSST*, vol. 4, 2004, pp. 283–300.

[13] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*. IEEE, 1999, pp. 182–189.

[14] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing Output Bottlenecks in a Supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 8.

[15] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The Case for Tiny Tasks in Compute Clusters," in *HotOS*, 2013.

[16] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI*, vol. 10, no. 1, 2010, p. 24.

[17] J. Liu, Y. Zhuang, and Y. Chen, "Hierarchical collective i/o scheduling for high-performance computing," *Big Data Research*, vol. 2, no. 3, pp. 117–126, 2015.

[18] D. Dai, Y. Chen, D. Kimpe, and R. Ross, "Two-Choice Randomized Dynamic I/O Scheduler for Object Storage Systems," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 2014, pp. 635–646.

[19] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-Side I/O Coordination for Parallel File Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 17.

[20] A. W. Marshall and I. Olkin, "A Multivariate Exponential Distribution," *Journal of the American Statistical Association*, vol. 62, no. 317, pp. 30–44, 1967.

[21] R. B. Ross, R. Thakur *et al.*, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th annual Linux showcase and conference*, 2000, pp. 391–430.

[22] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *FAST*, vol. 2, 2002, pp. 231–244.

[23] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[24] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the io performance of petascale storage systems," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–12.

[25] M. Dorier, G. Antoniu, R. Ross, D. Kimpe *et al.*, "CALCioM: Mitigating I/O Interference in HPC Systems Through Cross-Application Coordination," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 155–164.

[26] Y. Lu, Y. Chen, R. Latham, and Y. Zhuang, "Revealing applications' access pattern in collective i/o for cache management," in *The 28th International Conference on Supercomputing (ICS'14)*, 2014.

[27] D. Dai, Y. Chen, D. Kimpe, and R. Ross, "Provenance-based object storage prediction scheme for scientific big data applications," in *The 2014 IEEE International Conference on Big Data, (BigData'14)*, 2014.

[28] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/o acceleration with pattern detection," in *Proceedings of the 22nd international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2013, pp. 25–36.

[29] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 69–84.

[30] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Why Let Resources Idle? Aggressive Cloning of Jobs With Dolly," in *Presented as part of the*, 2012.

[31] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: Trimming Stragglers in Approximation Analytics," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 289–302.

[32] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[33] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 185–198.

[34] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI*, vol. 8, no. 4, 2008, p. 7.

[35] H. Xu and W. C. Lau, "Task-cloning Algorithms in a MapReduce Cluster With Competitive Performance Bounds," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*. IEEE, 2015, pp. 339–348.

[36] Q. Chen, C. Liu, and Z. Xiao, "Improving MapReduce Performance Using Smart Speculative Execution Strategy," *Computers, IEEE Transactions on*, vol. 63, no. 4, pp. 954–967, 2014.

[37] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[38] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 379–392.

[39] X. Ren, "Speculation-ware Resource Allocation for Cluster Schedulers," Ph.D. dissertation, California Institute of Technology, 2015.