Two-Choice Randomized Dynamic I/O Scheduler for Object Storage Systems

Dong Dai[1], Yong Chen[1], Dries Kimpe[2], and Robert Ross[2]

[1]Computer Science Department, Texas Tech University, USA, {dong.dai, yong.chen}@ttu.edu
[2]Mathematics and Computer Science Division, Argonne National Laboratory, USA, {dkimpe, rross}@mcs.anl.gov

***Abstract*—Object storage is considered a promising solution for next-generation (exascale) high-performance computing platform because of its flexible and high-performance object interface. However, delivering high burst-write throughput is still a critical challenge. Although deploying more storage servers can potentially provide higher throughput, it can be ineffective because the burst-write throughput can be limited by a small number of *stragglers* (storage servers that are occasionally slower than others). In this paper, we propose a *two-choice randomized dynamic I/O scheduler* that schedules the concurrent burst-write operations in a balanced way to avoid stragglers and hence achieve high throughput. The contributions in this study are threefold. First, we propose a two-choice randomized dynamic I/O scheduler with *collaborative probe* and *preassign* strategies. Second, we design and implement a *redirect table* and *metadata maintainer* to address the metadata management challenge introduced by dynamic I/O scheduling. Third, we evaluate the proposed scheduler with both simulation tests and experimental tests in an HPC cluster. The evaluation results confirm the scalability and performance benefits of the proposed I/O scheduler.**

I. Introduction

The object-based storage architecture promises improved manageability, scalability, and performance [1, 2]. It has been widely accepted in various parallel file systems (PFS) as a foundation of next-generation exascale platforms [3, 4]. In these object-based storage systems, files are arranged as a set of objects that are physically distributed into large numbers of object-storage devices (OSDs), which combine and leverage the CPU, network interface, and local cache with underlying disks or RAID. These object-storage devices take care of local data store tasks and leave the parallel file system to concentrate on data management jobs, such as metadata management, failure recovery, and user access control.

Typically, applications running under these object-based storage systems follow a standard "I/O-and-sync" pattern: first, each process scatters/gathers data to/from multiple OSDs concurrently; then they synchronize to make sure all the I/O operations have finished before entering the computation phase [5]. This ability of aggregating data from multiple storage servers provides improved I/O throughput [6]. In practice, however, at large scale the aggregation benefit is limited because of the increasing synchronization costs among more storage servers. One of the most critical reasons is the presence of *stragglers*, the slow servers during service of parallel I/O requests [7, 8].

Figure 1 shows two common cases of typical I/O access patterns. Each line represents an I/O request to a specific object storage server. The longer lines represent the slower storage servers (stragglers). The left figure shows a common scenario in which each process issues multiphase I/O requests and with a global synchronization. The right figure shows another common scenario when collective I/O is used [6], where processes synchronize after each I/O operation in order to reuse intermediate buffer. As we can see, for both scenarios, if any process hits a straggler, all the other parallel processes need to wait for the slowest operation to compute and hence waste valuable computation resources. This situation is expected to be significantly worse in the upcoming exascale era due to increased concurrency. As the DOE 2011 exascale report [9] shows, the bandwidth requirement in future platforms will be significantly larger (10 to 30x). Such a high bandwidth will likely require more storage servers to serve requests. When more storage servers are involved, the chance of having one straggler servicing parallel I/O requests substantially grows, which can drag down the overall performance significantly.

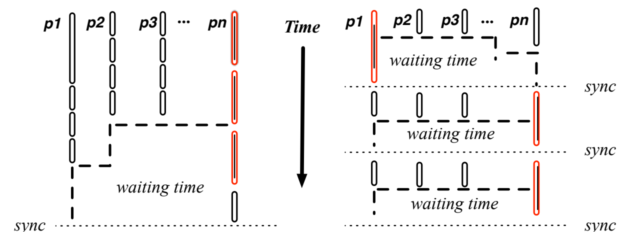


Fig. 1. Parallel I/O access with synchronization

To deliver high I/O bandwidth, we need to find a way to extend the scalability of aggregation bandwidth, in this case, by minimizing the number of I/O operations that hit the stragglers. Because of the urgent requirements of large-scale burst-writes from applications, in this paper our main target is minimizing the time to complete write operations involving stragglers. This is a hard problem because stragglers always exist in production HPC environments. Long-term stragglers, which can be slow for hours, days, or even always, may be caused by software bugs, hardware failures, or outdated hardware. Such long-term stragglers can possibly be detected and fixed by periodically monitoring the system. In contrast,

SC14, November 16-21, 2014, New Orleans, Louisiana, USA
978-1-4799-5500-8/14/$31.00

©2014 IEEE

short-term stragglers usually last only for minutes, seconds, or even less, making them harder to detect. Short-term stragglers may come from interference between applications or from resource contention between applications and system components [10]. For example, failure recovery in the storage server will significantly affect its performance. Since this type of straggler lasts only a short time, statistically it may still present an acceptable average performance. However, an individual request could be delayed significantly by them. In practice, these short-term stragglers are hard to detect and avoid.

In this paper, we propose a *two-choice randomized I/O scheduler*, which dynamically places write operations in different storage servers in a decentralized way. By tracking the real-time performance of storage servers, the scheduler is able to avoid most of the short-term stragglers and improve the parallel I/O performance significantly. The contributions of this work are threefold:

- Propose a two-choice randomized I/O scheduler (Section III), based on the analysis of the critical straggler problem in the HPC environment (Section II). With the idea of a two-choice randomized algorithm, we further introduce a *collaborative probe* and a *preassign* algorithm to further improve the scheduler performance (Section III).
- Design and implement a *redirect table* and a *metadata maintainer* to solve the metadata management challenge introduced by dynamic I/O scheduling (Section IV).
- Implement the proposed solution in an object-based parallel file system prototype (Triton [11]) and conduct evaluations (Sections V and VI). The performance and scalability results obtained by extensive evaluations are reported.

II. Background and Challenges

Avoiding short-term stragglers during intensive burst-writes situations is challenging. Below we discuss two of these challenges in more detail.

A. I/O Scheduler Challenge

Developing a proper scheduling algorithm itself is a challenge in an HPC environment. First, the algorithm must deal with *O(100K-1M)* parallel write processes from *O(1K-10K)* compute nodes to *O(1K)* storage servers, which is a common case in upcoming exascale HPC environment. This clearly requires a highly scalable scheduling algorithm and a distributed scheduler architecture, since centralized schedulers could otherwise easily become a bottleneck.

Second, an I/O scheduler must make decisions fast. Statistics show that many read/write requests are fairly small and complete relatively quickly [12]. Hence, it makes no sense to spend a significant amount of time on scheduling. In addition, the massive number of I/O requests also requires the scheduler to respond quickly.

Third, since the main aim of dynamic scheduling is to avoid the short-term stragglers that exist only for a short period of time, the scheduler must make scheduling decisions based on real-time performance data. Statistical or historical information will not be relevant. This issue is also another reason why centralized schedulers will not work, since they cannot know the accurate real-time status of the entire system without being a bottleneck.

B. Metadata Management Challenge

Before describing the metadata management challenge, we briefly introduce how metadata is managed in existing object-based parallel file systems. Currently, many parallel file systems have been designed and implemented by using an object storage model [13, 14, 15, 16, 17, 11]. The typical parallel file systems' metadata management strategies involve two steps in general: mapping files to objects and then mapping these objects onto object storage servers.

In the first step, most file systems split files into a set of objects. For example, Lustre [13], PanFS [14, 15], and PVFS [17] split their files into equal-sized stripes, which will be collected into objects on storage servers. In this way, file systems can easily calculate the list of stripes for each file by knowing only the stripe size and set of objects. On the other hand, file systems such as Ceph [16] distribute files into flexible-sized stripes, and the file system itself records them for each file.

The second step distributes objects onto different storage servers. File systems such as Lustre and PVFS follow a strict strategy to place stripes in a round-robin way starting from a given index. Ceph provides a CRUSH [18] algorithm to calculate a deterministic location for each object. File systems such as PanFS provide a flexible location strategy to store file units, and allocate extra objects to store this location information.

In summary, object-based storage systems need to choose how to split files and place the objects. After the objects are assigned to a storage server, any future update in that object will be delivered to that specific server. This approach helps parallel file systems avoid a heavy metadata management burden, but it also introduces a high possibility of hitting short-term stragglers. To avoid these stragglers, an I/O scheduler needs the ability to redirect writes to other servers and read them back later. However, this can severely complicate metadata management.

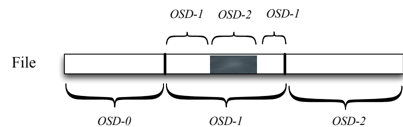


Fig. 2. Metadata fragments of dynamic scheduler.

In Fig. 2, the dynamic scheduler assigns a write request (the dark region) to server *OSD-2* instead of *OSD-1*, where its whole object is stored. In this case, the metadata manager will need to record the location of that fragment. As the number of redirected fragments increases, a severe burden is placed on the metadata manager. Another downside of changing metadata is that it requires invalidating client-side metadata caches. Clients, which outnumber of metadata servers, have

to recheck the metadata servers before each read, causing the metadata servers to become a bottleneck. Therefore, to efficiently implement an I/O scheduler, we need to find a way to efficiently manage metadata fragments. We defer this discussion until Section IV.

III. Two-Choice I/O Scheduler

To address the I/O scheduler and scheduling algorithm challenge, in this study we introduce *the power of two choices technique* [19, 20] as the foundation of our I/O scheduler and scheduling algorithm. The technique can be described as follows. For each I/O request, the scheduler randomly chooses two storage servers, determines their load, and sends the request to the storage server with the lower load. The "two-choice" strategy belongs to the randomized scheduler family, making it highly scalable since all schedulers can work concurrently. The two-choice strategy is an improvement over the purely random strategy which randomly chooses *one* storage server among all possible servers. Theoretical analysis [20, 21] shows that having two choices yields a qualitatively different type of behavior from the pure random strategy, leading to an exponential improvement.[1] However, because of the unique environment posed by HPC I/O systems, the purely two-choice randomized scheduling cannot be directly applied and is far from achieving the optimal result, as discussed in detail below.

A. Issues of Native Two-Choice Scheduler

Figure 3 plots the simulation results for an HPC cluster with 1,000 storage nodes and 10,000 concurrent processes, with each issuing synchronized 1 MB writes with different scheduling strategies. The response time is determined by the slowest process. We run the simulation multiple times and calculate the average response time for each case. In order to simulate the shared environment accurately, before parallel processes issue writes, pre-existing I/O loads are simulated on all storage servers. The I/O loads are generated based on a normal distribution with a small variation, which indicates an even and homogeneous cluster environment. To show how different scheduling strategies work with stragglers existing, we varied the straggler ratio in the storage servers from 0.0 to 0.5 (stragglers are simulated with 5 times more load). The round-trip time from compute nodes to storage nodes was set to be 5 ms.

The line *Fixed Scheduler* in this figure represents the traditional scheduler with round-robin data placement in current parallel file systems. This scheduler has a high possibility of leading to some I/O requests falling into straggler servers, which can in turn lead to a significant performance degradation. The line *Random Selection* represents a simple strategy of placing data into random storage servers. The line *Two-Choice Random* represents the case of selecting the better server from the two random choices of storage servers as we have described. Intuitively, the fixed scheduler could be the worst case because it always encounters the straggler (due to round-robin data distribution). However, the evaluations show different results. First, the purely random selection strategy can perform even worse than the fixed strategy. Second, the native two-choice randomized scheduler is not significantly better than the fixed strategy, even when the straggler number is small (fewer stragglers mean that the two-choice strategy can almost guarantee not hitting the straggler, whereas the fixed strategy hits stragglers).

[1] In fact, if we use *maximal load* as the maximum number of I/O requests distributed in one storage server to describe the imbalance of the scheduling algorithm, pure random strategy achieves a *maximum load* as $\frac{logn}{loglogn}$; however, the two-choice randomized algorithm provides a *maximum load* as $\frac{loglogn}{log2}$ with high probability [21, 20].

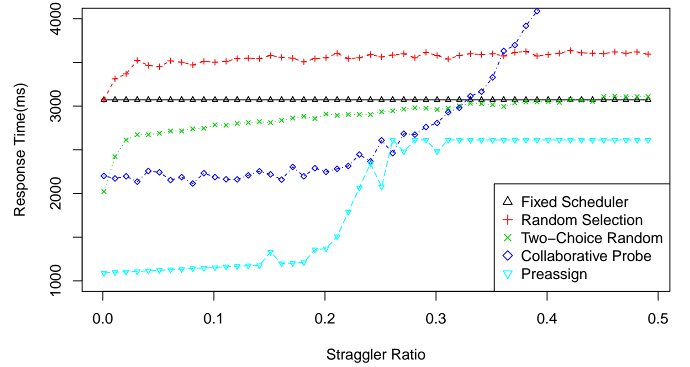


Fig. 3. Write response time of different strategies.

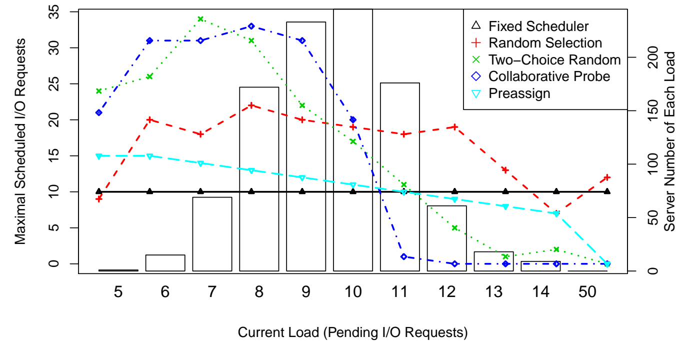


Fig. 4. Scheduling results of different schedulers.

To explain the reason for the "unexpected" performance and to direct our scheduler algorithm design, in Fig. 4 we further plot the actual scheduled I/O requests on each server by different schedulers. Since 1,000 storage servers are simulated, we aggregate them based on their loads. The x-axis shows different server loads. These bars, with a value reading from the *right* y-axis, represent the number of servers for each load. As we generated I/O loads for each server following a normal distribution, we can easily see that the number of servers for different loads fits this normal distribution as well. The rightmost bar represents the stragglers that were assigned with more loads. Its actual number is less than 10 (0.01%) in this simulation.

Figure 4 also uses different lines to represent the actual scheduled results of five different schedulers. These lines, with

a value reading from the *left* y-axis, represent the *maximum* received I/O requests of a server among all the servers with the same load (as we aggregate storage severs based on the load). For example, about 60 storage servers have loads of 7 in our simulation. After I/O requests are scheduled, some of these 60 servers may receive only 10 requests, but some may receive 20 requests. In this case, we plot the maximum value (20) because it shows the maximum write time of all the servers with the same loads, and the maximum time will determine the overall write performance. The schedulers that can avoid stragglers should have the maximum scheduled requests (a reading from the left y-axis) equal to 0 on the stragglers (the rightmost bar). Figure 4 shows that the fixed and random schedulers do not satisfy this requirement.

From Fig. 4, we can see that the fixed scheduler will always place the same number of I/O requests ($average = \frac{ioreq}{server}$) on each server, producing the most balanced I/O placement. The random selection distributes the requests to all servers evenly too, but the maximum number of scheduled requests is consistently greater than that of the fixed scheduler. Statistically, the random strategy places the $average$ I/O requests on each server, but there are variations due to the randomization; thus some servers receive more requests than the average, and some receive less. In addition, the random placement has a high possibility of hitting the stragglers and placing more writes than $average$ on one of these stragglers, leading to a consistently worse response time than the fixed scheduler, as revealed in Fig. 3.

Figure 4 also explains why the native two-choice random selection does not provide obvious better performance than that of the fixed scheduler even when it successfully avoids stragglers. In fact, the native two-choice random selection tends to place I/O requests to the servers with lower loads, thus leading to an unbalanced placement. As Fig. 4 shows, too many new write requests were scheduled to the servers with slightly lower loads, and in turn they become *new stragglers*. This unbalanced placement comes from the high concurrency nature of HPC applications, which could contain millions of concurrent processes. The huge number of I/O requests from these concurrent processes can arrive at the scheduler at nearly the same time, so all the schedulers will probe the storage servers concurrently. They will get the same load results and make the same scheduling decision, thus leading to new straggler servers. Another observation we can make from Fig. 3 is that when the number of stragglers increases, the response time of two-choice random selection increases, which shows that the native two-choice strategy still suffers from hitting stragglers.

From these extensive simulation results, we conclude that a desired I/O scheduler should both *avoid the existing stragglers effectively* and coordinate with each other to *avoid generating any new stragglers* while scheduling large amounts of parallel writes. In this study, we introduce the *collaborative probe* and *preassign* strategies to solve these problems. We also simulated these two strategies, and the results are reported in Fig. 3, and we can see that these two strategies significantly improve the response time for parallel write requests.

B. Collaborative Probe Strategy

The collaborative probe strategy comprises two steps. First, it combines concurrent probing in a single compute node together instead of probing separately. Second, it coordinates the scheduler and storage servers to gather more information for scheduling during the same probing time.

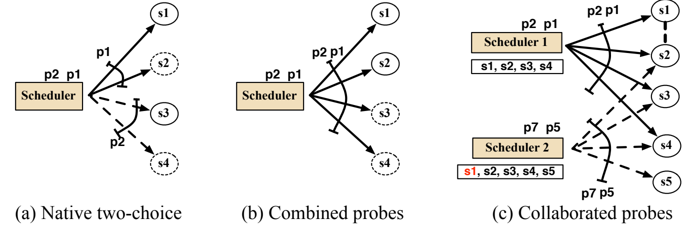


Fig. 5. Coordination between scheduler instances and storage servers.

In HPC, multiple I/O requests from different processes can be scheduled concurrently in most cases, such as the concurrent processes (*p1, p2*) pending in one scheduler, as Fig. 5 shows. The native two-choice algorithm probes two random servers for each I/O request. It will issue *k* probings and each for two storage servers for *k* concurrent I/O requests. Since there are only two choices, when too many I/O requests happen at nearly the same time, these two choices are likely both to be stragglers. Using collaborative probe, the scheduler combines all the probings for the pending requests, issues only one probing for *2*k* storage servers for *k* concurrent I/O requests, and selects *k* of them with least loads, as shown in Fig. 5(b). One can easily see that the possibility of choosing *k+1* stragglers in *2*k* is much smaller. Thus, collaborative probe and coordination among parallel I/O requests improve the response time.

The collaborative probe also enables the coordination between client-side scheduler instances and storage servers, as Fig. 5(c) shows. In this figure, *scheduler 1* and *scheduler 2* both have multiple (*k*) I/O requests pending, so they will both issue *2*k* probes instead of probing k times. Traditionally, during these *2*k* probes, the storage servers (*s1, s2, s3,...*) will tell the scheduler their current loads. Then, the scheduler will make decisions and send requests to the storage servers that are chosen as targets. There is no collaboration between schedulers and storage servers. In our design, the I/O scheduler tells the storage servers all the loads of probed servers in addition to the scheduling decision. This additional payload on requests is small and adds little overhead, but it provides a way for the storage servers to know the accurate load of other servers. As Fig. 5(c) shows, *s2* will be able to know the loads of *s1* after *scheduler 1* probes. When schedulers probe, this additional load data will be passed to the scheduler and provide more options for I/O requests. We note that the lifetime of the load information cached in each storage server could be larger in order to aggressively cache the historical information; or smaller to preserve accuracy. In the current

implementation, we set a very small time window (hundreds of milliseconds), which easily guarantees the accuracy and still benefits the performance significantly.

The simulation has confirmed that the *collaborative probe* improves the native two-choice randomized scheduler (Fig. 3). However, it still suffers the issue of possibly creating new stragglers, similar to the native two-choice algorithm. In fact, with collaborative probe, the problem of possibly creating new stragglers becomes even worse because each scheduler chooses *2*k* servers and places I/O requests to the least *k* servers. The storage servers with lighter loads have a much higher possibility of being chosen than does the native two-choice strategy. Figure 4 confirms this issue. After too many I/O requests are scheduled to the servers with lighter loads, that become the new stragglers and perform poorly. Next we further introduce a *preassign* strategy into the two-choice randomized algorithm to solve this problem.

C. Preassign Strategy

During scheduling, the storage servers reply to a scheduler with their current loads, and the scheduler compares all the collected loads and delivers the I/O requests to selected servers. There is a delay between probes and the time that the actual I/O requests arrive at the storage servers, and the server loads change only after I/O requests arrive. Therefore, any probes during this delay get loads that do not reflect requests about to be scheduled, which can cause schedulers to place too many I/O requests on the server with lighter loads. This problem is more significant in our case because of the highly concurrent I/O behaviors of HPC applications. To avoid this situation, the scheduler needs to know whether the recent probes to one server will assign I/O requests back to change its future load. In practice, the scheduler cannot know that, because it happens in the future; but we can use the *preassign* strategy to make an educated and accurate guess.

After a period of running, the storage servers will begin to understand what value of local loads are likely to be selected for servicing an I/O request. By maintaining the local history of selection decisions, the storage servers maintain a probability of acceptance for different loads values ($P(loads)$). With these values, the *preassign strategy* works as follows. Whenever a probing request arrives at the storage server, the server will collect local loads and reply. After replying, the storage server itself calculates the probability of being assigned. Before the scheduler makes the scheduling decision, the storage servers will preassign this I/O request to the local server based on the history information. Since it is not possible to know exactly the scheduling decision before the scheduler returns, the storage server updates its local loads l' based on the expectation $l' = l + P(loads) * len_{io}$, where $P(loads)$ denotes the possibility of being assigned given current loads and len_{io} is the length of current I/O request that probes. Once the scheduler responds and informs the storage servers whether they are chosen for servicing the request or not, the storage servers will adjust the preassigned loads. When the actual I/O requests are delivered to the selected servers, the accurate local loads will be updated.

We have simulated the effect of the preassign strategy as well. From Fig. 3 we can observe that the response times of parallel I/O requests were reduced significantly with the preassign strategy. It avoids most of the scheduling-generated stragglers and provides a stable performance even when the number of stragglers increases. Compared with the existing and common fixed scheduling strategy, it makes stable and significantly better scheduling decisions.

IV. Metadata Management

To address the metadata management challenge introduced by the dynamic I/O scheduler, we introduce two components: a *redirect table* and a *metadata maintainer*. The core idea of our solution is to keep the metadata strategies of the file systems *unchanged*. The file system's metadata management will always consider all objects to be stored at the OSDs where they should be, but the real location of each fragment is temporarily stored by the *redirect table* in each OSD. During idle time, relocated data will be moved back to keep the original file systems' data organization and clean up the redirect table.

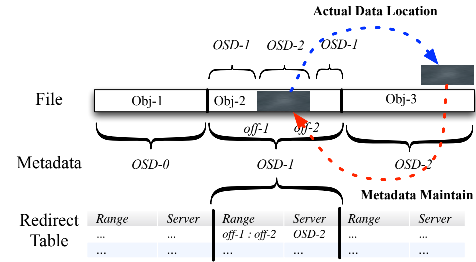


Fig. 6. Metadata of file and the redirect table.

Figure 6 shows how this redirect table works. A data fragment from *off-1* to *off-2* that the file system would place on OSD-1 is scheduled to OSD-2, and the actual data is stored there. After scheduling, a new entry will be created atomically in the redirect table of *OSD-1* denoting that the range between [*off-1, off-2*] of *Obj-2* is placed in OSD-2. The red dashed line shows the metadata maintainer thread running in the background on each storage server. It will move the data fragment in OSD-2 back to OSD-1 during idle time and remove the corresponding entry in the redirect table.

Using the redirect table and metadata maintainer instead of directly modifying the metadata provides several advantages. First, the metadata management burden will be evenly distributed to all the OSDs in the redirect table. Each object storage server takes charge of all the redirect information for the objects that ought to be stored there. As long as the object's distribution is balanced, the load on each OSD should be balanced. Second, all the modifications on the overlapped range of objects are directed to the same OSD and can be properly handled locally. This approach avoids the need for distributed transactions on the metadata management

and provides an easy implementation. Third, the redirect table does not need to change the default metadata management behaviors, which usually have been deeply optimized for each file systems. The metadata stored in file systems will be the same as in an unmodified system.

In the next section, we describe the detailed design and implementation of the redirect table, metadata maintainer, and other components to efficiently build the two-choice randomized dynamic I/O scheduler.

V. Design and Implementation

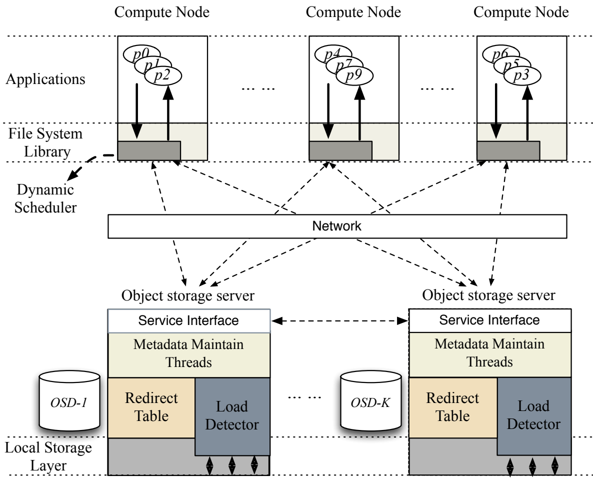


Fig. 7. Architecture of dynamic scheduler.

The software architecture of the proposed two-choice randomized dynamic I/O scheduler for object storage systems is illustrated in Fig. 7. On the compute server side, the client-side library is modified with an additional *dynamic scheduler* component. I/O requests are striped, merged, or collected into multiple requests to different OSDs by the original client library. These requests are dispatched by the dynamic scheduler that directs requests into different OSDs based on the two-choice randomized algorithm.

On the storage server side, there are three new components between the existing *service interface* and the *local storage layer*. A *Load Detector* collects local I/O loads and responds to the probing requests from the scheduler instances. It is tightly integrated with the local storage implementation as it needs to collect the runtime loads information in local servers. A *Redirect Table* works as a temporary distributed metadata management as we have described: it stores the redirect information for each non-local I/O fragment and keeps it persistent using the underlaying local storage. *Metadata Maintainer Threads* are a group of threads running on each object storage server. They continuously pull the scheduled data back into local storage and remove corresponding entries in the redirect table. The *Service Interface* is also enriched to serve the communication between storage servers.

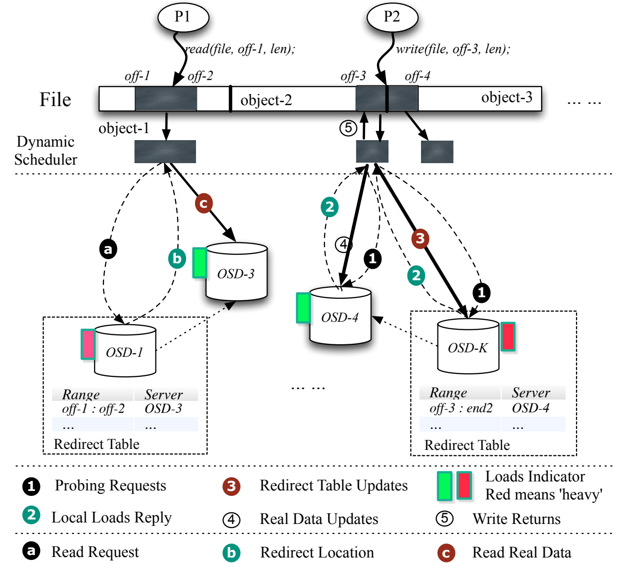


Fig. 8. Write/read scheduling procedure.

A. Scheduling Procedure

Figure 8 illustrates the schedule procedure through an example where two processes (*P1, P2*) running on different compute nodes try to read and write part of a file that has been striped into several objects. *P1* reads a part from *off-1* to *off-2*. *P2* writes a part from *off-3* to *off-4*, which happens to cross an object boundary. The write request from *P2* will be split into two parts by the client library since it contains data for two objects. Each of them will be distributed to one OSD. We take the first part as an example. These requests are sent to the dynamic scheduler, as Fig. 7 shows. Based on the two-choice randomized algorithm, the dynamic scheduler will choose two OSDs as the potential targets: one is *OSD-K* where the entire *object-2* is stored, and the other is *OSD-4* in this example.

After having two choices, the scheduler will send probing requests to both of them in *step 1*. OSD servers will reply to the scheduler with their local loads in *step 2*. This load information is collected by the *Loads Detector* component. In Fig. 8, *OSD-4* has a lighter load than *OSD-K* and should be selected as the final target. Thus, in *step 3*, the scheduler first sends the actual data into *OSD-4*. After the write returns successfully, it then sends the redirect table to *OSD-K* to update its redirect table, as *step 4* shows. Only if both of them succeed, the applications receive a successful return, as *step 5* shows.

The read request from process *P1* is also delivered to the dynamic scheduler. The scheduler sends the request to OSD according to the cached metadata, as step (a) shows in Fig. 8. It is possible that the actual data is scheduled to be written on another OSD in previous writes. The *OSD-1* will check its local redirect table first to see whether the requested data is stored elsewhere. Then it acknowledges the read and returns the actual data location to the scheduler in step (b). After

receiving the actual location, the scheduler will contact *OSD-3* and pull data for the client application.

The dynamic scheduler introduces latency for both read and write operations. During writes, the scheduler needs to probe and select the faster server before writing the real data. This adds a probing latency. Moreover, if the scheduler redirects the write, it introduces another latency in updating the redirect table. For reads, the scheduler needs to query the redirect table for the correct location before reading the real data. However, these extra latencies are actually small, and we argue that they are worthwhile considering the benefits of avoiding stragglers for intensive burst writes (and given the fact that stragglers will nearly always exist with the scale of the current and next-generation HPC). In the evaluation section, we report and discuss the results.

B. Redirect Table

The redirect table turns out to be one of the most important components in the proposed scheduler. We need to create a new entry on it (*create*), query whether a range has been redirected (*query*), and delete entries after metadata maintainer threads pull data back (*delete*). From the write/read scheduling procedure described in the previous subsection, we can see that the performance of these operations is critical: the longer they take, the more latency is introduced into I/O operations. Besides the performance consideration, we need to guarantee that the redirect table keeps a consistent status while concurrent operations happen, for example, *querying* a range while *create* and *delete* are running. Fault tolerance is also a consideration; the redirect table should be able to recover if the server failed and was restored.

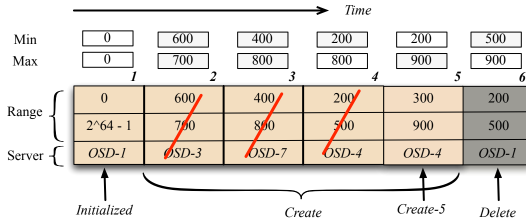


Fig. 9. Redirect table for object-1 in OSD-1.

The redirect table is implemented in a log-based structure. Each OSD keeps a log structure that stores all the *create* and *delete* operations for each object in a chronological order, as Fig. 9 shows. In each log record, it keeps the range and the target server. This log structure is created for each object (maximal size is $2^{64} - 1$ bytes) with an initialized record. The range of this record is [0, $2^{64} - 1$], and the target server is the local server. This setting represents that all data is stored locally at the beginning.

Whenever a new redirect decision is made, the scheduler will create a new entry in the redirect table; that is, a new log entry is appended at the end of the log structure for that object. If there are concurrent *creates* from different clients, the redirect table will order them based on the time the request is received. In such cases, all the operations are ordered in the OSD side. Whenever a new entry with range $[r_s,\ r_e]$ is appended in the log structure, all log entries before that time are checked. For any entry whose range is inside current range (inside means that the range start is equal to or larger than r_s and the range end is equal to or smaller than r_e), that entry is removed. As shown in Fig. 9, the second and third entries will be removed when entry (300, 900, *OSD-4*) is added. This design and implementation significantly save the search space for future *query* operations.

During the idle time, the metadata maintainer threads will go through the redirect logs one by one for all the objects and try to pull data back. It will issue a *delete* operation on the log entry whose data has been pulled back. In our current implementation, instead of removing the entry from the redirect table, a new entry with the same range but different target (local server) will be created and appended to the log structure. As we have described, this *create* operation will search and remove previous entries that fall into the new range, which in this case is exactly the one that should be deleted. For example, the fourth entry in Fig. 9 will be removed by the sixth *delete* operation.

During the process of *create* or *delete* operation, a Min/Max pair is also maintained, which denotes the largest range that may contain the redirection. This design is an optimization strategy for queries, and the algorithm is simple. For any range $[cr_s,\ cr_e]$ in the *create* operation, the Min value is updated if cr_s is less than the current Min, and the Max value is updated if cr_e is larger than the current Max. For range $[dr_s,\ dr_e]$ in a *delete* operation, only Min value is updated if dr_s is less than or equal to the current Min. In such cases, the Min is set to the dr_e value. In Fig. 9, we show all the modifications of Min/Max pair during creating and deleting on the redirect table. The gray boxes represent the value updated. Obtaining the Min/Max pair helps serve a not-inside query quickly.

Querying a range $[qr_s,\ qr_e]$ is simple. It starts from a timestamp, which should be the time the request is received, and looks back. If $[qr_s,\ qr_e]$ falls into the range of the current entry, that part is returned and the process continues with the left parts. For example, assuming we want to query [200, 600]'s location in Fig. 9, it is first compared with the sixth entry. If it is found the [200, 500] is actually stored locally (has been pulled back by metadata maintainer threads), that location is returned and the operation continues with [500, 600]. By comparing it with the fifth entry, it is found out that the range is stored on *OSD-4*; that location is returned. The query performance is determined by the length of in-memory log structure. The length is continuously reduced by removing entries during *create* and *delete* operations. An important advantage of the log-based structure is that it can be persisted as an append-only on-disk log files and can be easily rebuilt from log file during recovery. This design and implementation help improve the fault tolerance of metadata management.

C. Metadata Maintainer Threads

Metadata maintainer threads run in the background of each OSD to pull back data from other servers. The number of these threads is configurable. Several considerations arise in designing metadata maintainer threads. First, the thread starts only if both the local OSD and remote OSD are idle. This case is highly possible because of the bursty nature of I/O activities in an HPC environment [12]. Moreover, the running threads can be stopped during the process of pulling data from remote OSD.

Second, a consistency problem could arise if new redirect table entries are created while maintainer threads are pulling data. In Fig. 10, we show a typical example. The *Metadata Maintainer Thread 1* is working on a redirect entry (*r1, r2, OSD-1*) to pull the *DATA-1* inside OSD-1. During this process, a new scheduler decision is made to redirect the same data to OSD-2. As we have described, when *Thread 1* returns, it should add a new *delete* entry into the redirect table (as the dashed line shows). However, adding this new entry will invalidate the entry that points to OSD-2. Thus, the system will consider the data pulled from OSD-1 is the correct one, whereas the data in OSD-2 is correct. To solve this problem, after the metadata maintainer threads finish pulling, they will add the new *delete* entries with the timestamp when the thread is started, instead of the current time. For example, in Fig. 10, the *delete* entry will be inserted before (*r1, r2, OSD-2*). This design makes any redirection after the pulling thread still correct.

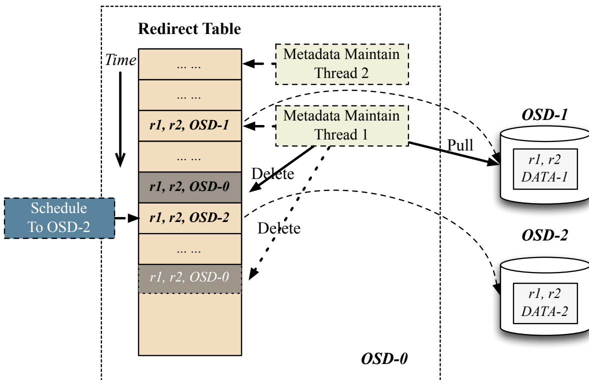


Fig. 10. Metadata maintainer threads consistency.

D. Load Detector

The *Load Detector* implementation is tightly coupled with the underlaying file systems. It needs to collect the runtime information of the local server especially the I/O load. Building an accurate model to predict the performance of specific I/O operations is still an open research question. In this paper, we use two metrics: the number of concurrent I/O operations and the size of concurrent I/O operations to represent the local load. Clearly, other metrics can be used, but they are not the main focus of this current study.

VI. Evaluation

To validate the design and evaluate the performance of the proposed I/O scheduler, we first conducted micro-benchmark evaluations on the critical components including the collaborative probe and redirect table. Then, we evaluated the entire I/O scheduler integrated with a parallel file system (Triton [22]) running real-world workloads. These tests were conducted in addition to the simulation tests presented in Section III. We summarize two principal findings here based on experimental results:

- The proposed dynamic I/O scheduler introduces a small and stable latency for both reads and writes (less than 5 ms for each read operation, less than 10 ms for each write operation) in a scalable way.
- The two-choice randomized scheduler is able to avoid the short-term stragglers and provides much better write throughput compared with other strategies. Moreover, it generates well-balanced I/O loads on each storage server.

All evaluations in this section are based on the current implementation on Triton, an object-based storage system designed at Argonne National Laboratory (ANL) [11]. The evaluations were conducted on the Fusion cluster located in Argonne National Laboratory Computing Resource Center [23]. It contains 320 compute nodes. Each node has a dual-socket, quad-core 2.53 GHz Intel Xeon CPU with 36 GB memory and 250 GB local hard disk. All nodes are connected by high-speed network interconnect (InfiniBand QDR 4 GB/s per link, per direction). The global parallel file system includes a 90 TB GPFS file system and a 320 TB PVFS file system.

A. Evaluation with Micro-benchmarks

Before evaluating the newly proposed scheduler with real-world workloads, we tested two most critical components introduced by the proposed dynamic scheduler: the probing and redirect table components. They are on the critical path that all writes and parts of reads need to go through.

1) Probing Performance: Probing exists in almost every write operation: whenever an I/O request arrives at the scheduler, it will issue a parallel probing to random-chosen storage servers. With the *collaborative probing* strategy, if there are k concurrent I/O requests, the scheduler will issue *2*k* probes to storage servers. For a massively parallel I/O request, all scheduler instances (S_{ins}) may get involved, and producing $2*k*S_{ins}$ probes in total. On the other hand, since the storage servers are randomly selected, statistically each of them will receive approximately the same number of probing requests. Thus, the more storage servers we choose, the fewer the probings on each of them. In this evaluation, in order to show the performance and scalability of probing, the probing loads on each server are maximized by setting the number of storage servers equal to *2*k*. In this case, each scheduler on each node will probe all the storage servers. To further make sure all the probings happen concurrently on the storage server side, each scheduler repetitively probes all the servers 100 times.

Figure 11 shows the probing performance on Fusion: the x-axis shows the number of schedulers, in this case from 1

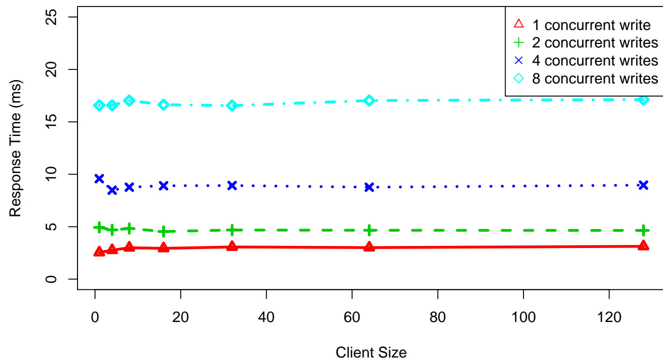


Fig. 11. Probing performance in different scenarios.

to 128; the y-axis shows the time cost of probing. Since the scheduler probes more than one server each time, we record only the slowest one as the time cost of that probing. Different lines show the performance for different k values. From the results plotted in this figure, we can make two conclusions. First, the response time of probing is stable for each case when the number of clients increased. This observation indicates that, in our evaluation setting, the storage servers will not become the bottleneck with more concurrent probing requests. Second, the response time of probing is minor: it costs less than 10 ms for four concurrent write requests. Assuming each write contains 2 MB data, and considering the write speed between one compute node and one storage node is limited to 50 MB/s in the current Fusion setting, this probing introduces about 6% delay, which is acceptable especially considering the performance benefits from avoiding stragglers. Table I shows the delay percentage with different concurrent I/O requests. For each write size, the percentage is stable and slightly reduced with heavier loads. From these tests, we observe that the proposed scheduler does not produce benefits for very small writes ($O(1B - 1KB)$) because the probing can cost a considerable amount of time. We solve this issue by providing a configurable minimal write size threshold (e.g., 1 KB) in the current implementation.

TABLE I
DELAY PERCENTAGE FOR DIFFERENT CONCURRENT WRITE SIZES.

Write Size	*1 Write*	*2 Writes*	*4 Writes*	*8 Writes*
1 MB	12.5%	11.7%	10.6%	10.3%
2 MB	6.25 %	5.85 %	5.3%	5.1%
4 MB	3.12%	2.93 %	2.15%	2.1%

2) Redirect Table Performance: Processing the redirect table introduces extra latency during I/O scheduling. First, the scheduler needs to create a new entry into the redirect table each time a data chunk has been stored in another storage server; thus the *create* performance matters for writes. Second, a local storage server needs to query its redirect table before any reads, so the *query* performance determines the read latency. Based on the design of the redirect table, we conclude that the complexity of both *create* and *query* operations is based on *O(n)*, where n represents the length of the redirect table. To evaluate its performance, we created a highly write-intensive workload with all processes issuing redirected writes in a random range of one object, each range belonging to [0, 100 MB], which is common as a single object size.

Figure 12 shows the redirect table length and the response time of *query, create* requests when more entries are being created. We observe that the length of the redirect table is kept small compared with the much greater number of operations. This observation confirms the superior *create* and *query* performance (with an almost guaranteed delay of less than 5 ms). Since the *metadata maintainer* threads will continuously move data back in the background, the redirect table will perform even better in a long run. Note that the only read latency introduced by the proposed dynamic I/O scheduler comes from the redirect query. Based on the results from these tests, we conclude that the cost of dynamic scheduling is affordable.

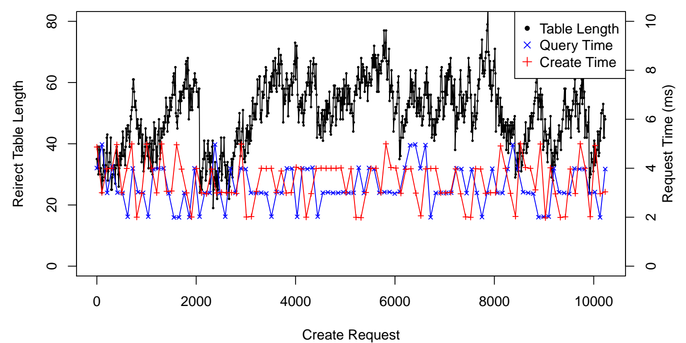


Fig. 12. Redirect table performance.

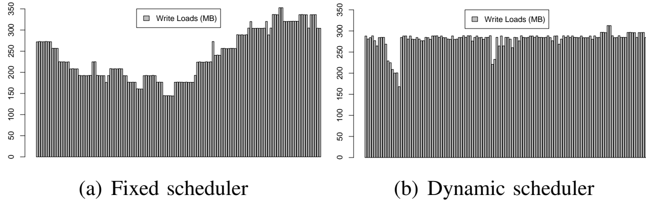
B. Evaluation with Workloads

To evaluate the proposed scheduler with real-world workloads, we first analyzed typical burst-write applications running on current cutting-edge HPC systems for their I/O patterns. Table II shows the three most write-intensive applications with production runs on the Intrepid IBM Blue Gene/P system in Argonne National Laboratory and their I/O behaviors [24]. These I/O behaviors were collected by the Darshan lightweight I/O characterization tool [25]. Not surprisingly, these data confirm that in a shared production system, many workloads have significantly different sizes and run at the same time. Furthermore, the write size is typically larger than 1 MB. Based on our previous evaluations, they will work well with the new I/O scheduler.

TABLE II
WRITE-INTENSIVE JOBS ON INTREPID [26]

Application Name	*Nodes*	*Write Size/Node*	*Times*
PlasmaPhysics	32,768	1.0 GB	1
AstroPhysics	8,096	≈50 MB	9
Turbulence2	4,096	59.0 MB	22

Based on these features, we created a group of I/O workloads to mimic the real-world applications' I/O behaviors. There are larger workloads, like PlasmaPhysics, that would



(a) Fixed scheduler (b) Dynamic scheduler

Fig. 13. Server loads while scheduling all the workloads.

take all the storage servers and clients to run, and 10 smaller applications, like AstroPhysics, that only use half of storage servers and clients. Another set of 10 applications, like Turbulence2, have even fewer resource requirements and take only one-fourth of the resources to run. Each workload writes to a different file, and each process writes 4 MB data each time. All the workloads concurrently share a cluster, which includes 128 storage servers and 64 compute nodes (512 CPUs).

First, we run all these workloads with the fixed round-robin scheduler, which reflects how they are executed in the current storage system. To avoid placing I/O requests in an imbalanced manner, we choose a random start index for each file. However, because the size of each workload is different, the total write requests on each storage server will not be even, as Fig. 13(a) shows. The x-axis shows all the storage servers, and the y-axis represents the write data size on each storage server. One can easily observe the imbalance among the servers. Then, we ran all these workloads again, using the new dynamic I/O scheduler. Fig. 13(b) shows how these I/O requests are evenly distributed. The overall execution time is reduced because the maximum write size is much smaller.

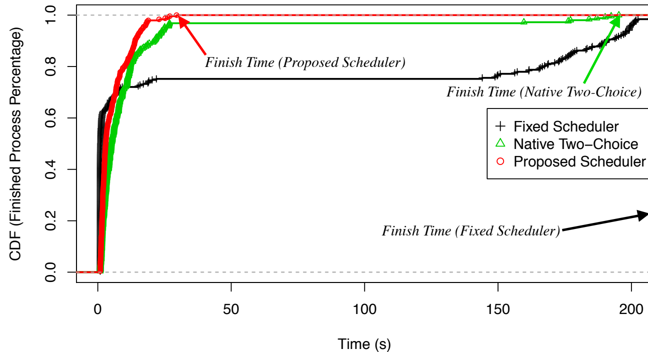


Fig. 14. *CDF of completion times for each process with different schedulers*. These results were generated on Fusion with 64 storage servers running and 512 clients issuing I/O requests. Each process issues 20 MB writes directly from local memory buffer.

We have shown that the proposed dynamic I/O scheduler improves the balance among different storage servers, and it also accelerates a single application when short-term stragglers exist. In this set of tests, we ran the same workloads in the background using the fixed I/O scheduler. The previous evaluation confirmed that these tests created imbalanced performance among storage servers, which indicates the existence of short-term stragglers. Then, we submitted a new workload that ran on all the client nodes issuing write requests. Figure. 14 shows the CDF (cumulative distribution function) of completion times for each process in this workload. From the results plotted in this figure, we observe that at the first several seconds, the fixed scheduler schedules more processes to finish. This phenomenon indicates that the fixed scheduler is placing fewer I/O workloads on the storage servers with lighter loads. For all the processes, however, the proposed two-choice randomized dynamic I/O scheduler provides a short finish time by avoiding stragglers (the red arrow in Fig. 14), whereas the fixed scheduler is slowed significantly by them (the black arrow). In addition, we also plot the performance of native two-choice algorithm. We can see there are processes that hit the stragglers and slow the whole application (the blue arrow). Many other test results also confirmed these observations and are omitted due to the page limit.

In summary, as confirmed by both simulation and experimental evaluations, the existence of storage server stragglers has a clear impact on the I/O system performance. The newly proposed two-choice randomized dynamic I/O scheduler provides a scalable, efficient, and fast solution that avoids stragglers and achieves considerably better overall performance.

VII. Related Work

Existing research investigated avoiding stragglers and evenly distributing I/O tasks. Client libraries such as ADIOS [27] provide a pure client-side I/O scheduler for parallel applications [28, 29]. Our work differs in that scheduler works as a file system component and is resilient to client-side failures. Moreover, distributed two-choice randomized strategy provides a better scalability than ADIOS's centralized approach. In addition to client-side schedulers, researchers have proposed server-side I/O schedulers which re-order I/O requests within each server to improve the overall system throughput [5]. Our work aims to optimize per-application throughput.

Sparrow [30] is another scheduler inspired by the two-choice algorithm. It was designed for scheduling a large number of short tasks in a cloud environment while avoiding stragglers. However, scheduling cloud tasks is vastly different from redirecting HPC I/O operations: the former does not need to track redirections, and the high concurrency nature of the latter makes existing strategies not work. In Hadoop-like systems, other strategies have been proposed to avoid stragglers [31, 32]. Our work differs in scheduling strategy as well as environment (highly consistent file systems).

QoS-aware I/O schedulers have also been extensively studied in storage systems [33, 34, 35, 36]. To keep QoS (quality of service) or SLA (service level agreement), these schedulers usually focus on resources provision or performance isolation among different clients and workloads. Comparingly, our proposed solution focuses more on avoiding short-term stragglers in order to improve the overall performance of storage systems, significantly departures from them. It can also be applicable to improve the QoS support by tweaking our proposed scheduler with proper resource provision and performance isolation.

According to previous research [37], resource provision can be done more efficiently because our proposed scheduler provides the real-time statuses of storage servers.

VIII. Conclusion and Future Work

In this paper, we present the idea, design, implementation, and evaluation of a *two-choice randomized dynamic I/O scheduler* for object-based storage systems. This work extends the native two-choice randomized algorithm to I/O schedulers and adds *collaborative probe* and *preassign* strategies. We design and implement the necessary components (*redirect table* and *metadata maintainer*) to solve the metadata challenge introduced by adding dynamic placement in parallel file systems. Our evaluation confirms both a better response time for applications and a better load balance for storage servers, with limited overhead because of load probing and metadata management. While our current solution improves I/O performance, we believe that even more insights can be extracted from the probed information, and we will investigate this aspect in future work.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357, and by the National Science Foundation under grant CNS-1338078 and CNS-1162488. We gratefully acknowledge the computing resources provided on "Fusion," a 320-node computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] 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.

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

[3] E. Barton, "DAOS containers: A storage abstraction for exascale," *presentation at Argonne National Laboratory*, Oct. 2011.

[4] D. Goodell, S. J. Kim, R. Latham, M. Kandemir, and R. Ross, "An Evolutionary Path to Object Storage Access," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 36–41.

[5] 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.

[6] 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.

[7] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing Output Bottlenecks in a Supercomputer," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–11.

[8] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O Performance Challenges at Leadership Scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 40.

[9] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, W. Bethel, H. Childs *et al.*, "Scientific Discovery at the Exascale: Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization," 2011.

[10] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. ACM, 2008, pp. 15–24.

[11] C. Karakoyunlu, D. Kimpe, P. Carns, K. Harms, R. Ross, and L. Ward, "Towards a Unified Object Storage Foundation for Scalable Storage Systems," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–8.

[12] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[13] P. Schwan, "Lustre: Building a File System for 1000-Node Clusters," in *Proceedings of the 2003 Linux Symposium*, vol. 2003, 2003.

[14] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas Activescale Storage Cluster: Delivering Scalable High Bandwidth Storage," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 53.

[15] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System." in *FAST*, vol. 8, 2008, pp. 1–17.

[16] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A Ccalable, High-Performance Distributed File System," in *Proceedings of the 7th symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 307–320.

[17] 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.

[18] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data," in *Proceedings of the 2006*

ACM/IEEE conference on Supercomputing. ACM, 2006, p. 122.

[19] M. Mitzenmacher, "On the Analysis of Randomized Load Balancing Schemes," *Theory of Computing Systems*, vol. 32, no. 3, pp. 361–386, 1999.

[20] M. a. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 10, pp. 1094–1104, 2001.

[21] M. Raab and A. Steger, "Balls into Bins: A Simple and Tight Analysis," in *Randomization and Approximation Techniques in Computer Science*. Springer, 1998, pp. 159–170.

[22] "Triton," in *http://www.mcs.anl.gov/projects/triton/*.

[23] "Fusion," in *http://www.lcrc.anl.gov/fusion/*.

[24] "Intrepid," in *https://www.alcf.anl.gov/intrepid*.

[25] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of Petascale I/O Workloads," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[26] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the Role of Burst Buffers in Leadership-Class Storage Systems," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–11.

[27] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield *et al.*, "Hello ADIOS: The Challenges and Lessons of Developing Leadership Class I/O Frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[28] 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.

[29] Q. Liu, N. Podhorszki, J. Logan, and S. Klasky, "Runtime I/O Re-Routing+ Throttling on HPC Storage," in *5th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX, 2013.

[30] 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.

[31] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones," in *Proceedings NSDI*, 2013.

[32] 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.

[33] J. C. Wu and S. A. Brandt, "Providing Quality of Service Support in Object-Based File System." in *MSST*, vol. 7, 2007, pp. 157–170.

[34] A. Gulati, A. Merchant, and P. J. Varman, "pClock: An Arrival Curve Based Approach For QoS Guarantees In Shared Storage Systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1, pp. 13–24, 2007.

[35] W. Jin, J. S. Chase, and J. Kaur, "Interposed Proportional Sharing for a Storage Service Utility," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1. ACM, 2004, pp. 37–48.

[36] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Façade: Virtual Storage Devices with Performance Guarantees." in *FAST*, vol. 3, 2003, pp. 131–144.

[37] L. Lu, P. Varman, and K. Doshi, "Graduated QoS By Decomposing Bursts: Don't Let The Tail Wag Your Server," in *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*. IEEE, 2009, pp. 12–21.