vPFS+: Managing I/O Performance for Diverse HPC Applications

Ming Zhao
Arizona State University

Yiqi Xu
VMware Inc.

Abstract*—High-performance computing (HPC) systems are increasingly shared by a variety of data- and metadata-intensive parallel applications. However, existing parallel file systems employed for HPC storage management are unable to differentiate the I/O requests from concurrent applications and meet their different performance requirements. Previous work, vPFS, provided a solution to this problem by virtualizing a parallel file system and enabling proportional-share bandwidth allocation to the applications; but it cannot handle the increasingly diverse applications in today's HPC environments, including those that have different sizes of I/Os and those that are metadata-intensive. This paper presents vPFS+ which builds upon the virtualization framework provided by vPFS but addresses its limitations in supporting diverse HPC applications. First, a new proportional-share I/O scheduler, SFQ(D)+, is created to allow applications with various I/O sizes and issue rates to share the storage with good application-level fairness and system-level utilization. Second, vPFS+ extends the scheduling to also include metadata I/Os and provides performance isolation to metadata-intensive applications. vPFS+ is prototyped on PVFS2, a widely used open-source parallel file system, and evaluated using a comprehensive set of representative HPC benchmarks and applications (IOR, NPB BTIO, WRF, and multi-md-test). The results confirm that the new SFQ(D)+ scheduler can provide significantly better performance isolation to applications with small, bursty I/Os than the traditional SFQ(D) scheduler (3.35 times better) and the native PVFS2 (8.25 times better) while still making efficient use of the storage. The results also show that vPFS+ can deliver near-perfect proportional sharing (>95% of the target sharing ratio) to metadata-intensive applications.

***Index Terms*—I/O scheduling, parallel storage, performance management**

I. Introduction

High-performance computing (HPC) systems remain to be indispensable for solving challenging computational problems in many disciplines. Such systems deliver high performance to applications through parallel computing on large numbers of processors and parallel I/Os on large numbers of storage devices. Applications are becoming increasingly data intensive in HPC systems. On one hand, the emergence of "big data" is fostering a rapidly growing number of data-driven applications which rely on the processing and analysis of large volumes of data. On the other hand, as applications employ more processors to solve larger and/or harder problems, they are forced to checkpoint more frequently in order to cope with the reduced mean time to failures. The implication of the proliferation of data-intensive applications is that I/O performance now plays a growing, crucial role to HPC systems.

At the same time, HPC applications are increasingly deployed onto shared computing and storage infrastructures. Similar to the motivations for cloud computing, consolidation brings significant economical benefits to both HPC users and providers. For data-intensive HPC applications, hosting popular data sets (e.g., human genome, weather data, digital sky survey, Large Hadron Collider experiment data) on shared infrastructure also allows these massive volumes of data to be conveniently and efficiently shared by different applications. Consequently, today's HPC systems are typically not for dedicated use by particular applications anymore; they are, instead, shared by applications with diverse resource demands and performance requirements.

It is the combination of the above two concurrent trends that makes resource management, particularly the management of shared storage resources, an important and challenging problem to HPC systems. Although the processors of an HPC system are relatively easy to partition in a space-sharing manner, the storage bandwidth is difficult to allocate because it has to be time-shared by applications with varying I/O demands. Without proper isolation of competing I/Os, an application's performance may degrade in unpredictable ways when under contention.

To address the need of I/O performance management in HPC systems, previous work studied vPFS, a virtualization-based approach to parallel file system I/O scheduling [30]. It employs user-space proxies to transparently interpose parallel file system I/Os and schedule them on a per-application basis. It then adopts a proportional-share scheduler, SFQ(D) [14], to schedule the reads and writes from competing applications and allow them to share the HPC system's total I/O bandwidth in a fair and work-conserving manner. However, vPFS has two key limitations—it cannot provide good performance isolation to applications with small I/Os and applications that are metadata-intensive. These limitations seriously hinder the support of the increasingly diverse applications. For example, many applications that access the HPC storage via the POSIX interface require large numbers of small I/Os, and many others that work with small files are inherently metadata-intensive [19].

This paper presents vPFS+, a new solution to provide I/O performance management to diverse HPC applications. It builds upon the virtualization framework provided by vPFS, and addresses the aforementioned limitations. First, vPFS+ employs a new proportional-share I/O scheduler, SFQ(D)+, which allows applications with various I/O sizes and issue rates to share the parallel storage with good application-level fairness and system-level utilization. It recognizes the limitation of the traditional SFQ(D) scheduler by considering the different costs of dispatched I/Os when they are processed by the underlying storage. It further employs a new backfill I/O scheduling technique to promote the dispatching of small I/Os and improve the storage utilization. Second, vPFS+ extends the scheduling to also cover metadata I/Os and optimizes it according to the characteristics of these operations. As a result, vPFS+ is able to achieve fair sharing of the entire parallel storage system's total data and metadata services for diverse HPC applications.

The vPFS+ approach can be applied to different parallel file systems and transparently deployed on existing HPC systems. A prototype is implemented on PVFS2 [8], an open-source parallel file system which is used both in production HPC systems and as a flexible research platform [1], [7], [21], [32]. It is evaluated using a comprehensive set of representative HPC benchmarks and applications, including the IOR [17] benchmark, BTIO from the NAS Parallel Benchmark suite [29], a real-world scientific application WRF [28], and a metadata benchmark multi-md-test from PVFS2 [8]. The results show that the new SFQ(D+) scheduler can provide significantly better performance isolation for applications with small, bursty I/Os (when it is under intensive I/O contention) than the traditional SFQ(D) scheduler (3.35 times better) and the native PVFS2 (8.25 times better) while still making efficient use of the storage (13.81 times better total throughput than a non-work-conserving scheduler). The results also show that the new SFQ(D)+ scheduler can achieve near-perfect proportional sharing for competing metadata-intensive applications ($> 95\%$ of the target sharing ratio). Finally, the overhead of vPFS+ is small.

The rest of this paper is organized as follows: Section II introduces background and motivations; Section III describes the SFQ(D)+ scheduler; Section IV discusses metadata scheduling; Section V presents the evaluation; Section VI examines the related work; and Section VII concludes this paper.

II. Background and Motivations

A. Parallel File System based HPC Storage

In a typical HPC system, applications run on the compute nodes and access their data stored on the storage nodes through a parallel file system. A parallel file system (e.g., Lustre [18], GPFS [23], PVFS2 [8], PanFS [27]) consists of clients, data servers, and metadata servers. The clients run on the compute nodes (or I/O nodes that perform I/Os on behalf of the compute nodes) and provide the interface (through POSIX or MPI-IO) to the parallel file system. Metadata servers are responsible for managing file naming, data location, and file locking. Data access typically has to first go through a metadata server to obtain the appropriate permission and the location on the corresponding data servers for the requested data. To avoid the metadata server from becoming a bottleneck, parallel file systems can distribute metadata management across several metadata servers. Finally, data servers run on the storage nodes and are responsible for performing reads and writes on the locally stored data. Each data request issued by a client is usually striped across multiple data servers to achieve high performance by serving the striped requests in parallel on their storage nodes.

In current HPC systems, the storage infrastructure is considered opaque by applications: it is shared by all the compute nodes and it serves applications' I/O demands in a best-effort manner. Although it is straightforward to partition the compute nodes (and their processors) among multiple concurrent applications, the parallel file system storage has to serve the concurrent I/O requests from all the applications that are running in the system, which often have distinct I/O access patterns and performance requirements. However, the parallel file system based storage is not designed to recognize the different I/O demands from applications—it sees only generic I/O requests arriving from the compute nodes. Neither is the storage system designed to satisfy the different performance requirements from applications—it is engineered to meet the maximum throughput target for the entire HPC system.

HPC applications in fact differ greatly in their data access patterns and storage bandwidth requirements. For example, mpiBLAST loads a large amount of genome data when it starts but not much I/O afterwards; WRF [28] generates many small I/Os for its input, output, and checkpoint data throughout the run; and S3D produce a large volume of restart files periodically in order to tolerate failures during its execution. Applications also have different priorities, e.g., due to different levels of urgency or business value, which should be reflected on the scheduling of their I/Os. For example, the execution of WRF for the forecast of an impending hurricane should be given the highest priority, but it may not be necessary to dedicate the entire system to WRF, as the obtainable speedup often does not scale to the system size. Therefore, applications with different storage demands and performance requirements are multiplexed on the shared storage system, which will become increasingly more common with the continued scale-up of HPC systems. Hence, per-application allocation of shared parallel storage service is key to delivering application-desired performance, which is generally lacking in existing HPC systems.

B. Virtualization-based Parallel I/O Scheduling

To address the need of I/O performance management in HPC systems, the previous work, vPFS [30], studied virtualization-based parallel I/O scheduling. The general strategy taken by vPFS is based on the *virtualization principles*, where an indirection layer exposes the parallel file system interfaces already in use by the storage system for I/O accesses. This strategy allows applications to time-share the I/O resources without modifications, while parallel I/O schedulers are placed upon the indirection layer to provide I/O performance management. To create this virtualization, vPFS employs I/O proxies between the shared native parallel file system clients and servers, which differentiate per-application

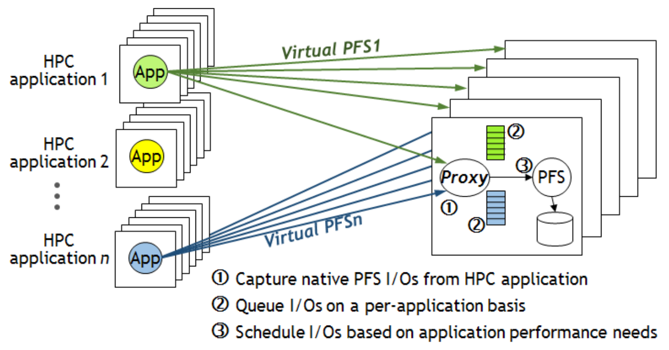

HPC application 1
App
HPC application 2
App
HPC application n
App
Virtual PFS1
Virtual PFSn
Proxy
PFS
① Capture native PFS I/Os from HPC application
② Queue I/Os on a per-application basis
③ Schedule I/Os based on application performance needs

Fig. 1: Architecture of virtualization-based parallel file system (PFS) I/O scheduling

I/Os and enforce their resource allocation. This proxy-based virtualization approach can be applied transparently to existing HPC system deployments with a small performance overhead. It can support different parallel file system protocols as long as the proxy understands the protocols and handles the I/Os accordingly.

Specifically in vPFS, the proxies are spawned on every parallel file system server to broker the applications' I/Os across the system, where the requests issued by a parallel file system client are first processed and queued by a proxy and later forwarded to the native parallel file system server according to the scheduling policy (Figure 1). To differentiate the I/Os from different applications, a proxy can use their source network addresses to identify their ownership, because HPC systems commonly partition compute nodes across concurrent applications so each node executes a single application's processes. In the case when multiple applications are run on the same compute node, each application's I/Os can be directed to a specific port of the proxy so that its I/Os can be uniquely identified using the source network address and port number combination. Each proxy employs a classic scheduler, SFQ(D) [14], to provide proportional sharing of the service available from its local data server. vPFS then provides several efficient global coordination schemes for the distributed schedulers to cooperate on global I/O scheduling and allow applications to share the system's total I/O service with good fairness and efficiency.

Based on this design, vPFS achieved good results in providing performance isolation among applications with large, intensive I/Os, e.g., it can provide near-perfect proportional bandwidth sharing (within 4% of the target sharing ratio) for two checkpointing applications [30]. However, it still has two important limitations. First, vPFS cannot handle applications with diverse I/O sizes. For example, the execution of WRF often generates a large number of small I/Os (on average 32KB in size), which are much smaller than the large checkpointing I/Os from other applications. When these applications run together, the application with small I/Os will be impacted severely as vPFS cannot handle such interference well. Second, vPFS does not support the scheduling of metadata I/Os, which are increasingly important to the performance of modern HPC applications. In fact, a recent I/O behavior analysis of multiple production HPC systems shows that 40% of the studied applications spend more time in metadata operations than reading/writing data [19]. Hence, the lack of support for metadata I/O scheduling will compromise the performance isolation even if the data I/Os can be well handled by vPFS.

To address the above limitations, this paper presents vPFS+ which includes 1) a new I/O scheduler, SFQ(+), to support applications with diverse I/O sizes, and 2) the new ability to schedule metadata I/Os and provide performance isolation to metadata-intensive applications. The rest of the paper presents the design, implementation, and evaluation of vPFS+ in these two aspects.

III. SFQ(D)+

A. Limitation of Classic SFQ Schedulers

Proportional sharing of parallel storage bandwidth is important to HPC applications because their performance targets are often specified in terms of turnaround times which depend on the shares of CPU cycles and I/O bandwidths that the applications can get from the system. It is relatively straightforward to allocate CPU shares based on cores and time slices, but it is non-trivial to allocate I/O bandwidth shares. Proportional-share I/O schedulers can provide this key missing control knob in HPC resource management. Proportional resource sharing is defined as when the total demand is greater than the available resource, each application should get a *share* of the resource proportionally to its assigned *weight*. In an HPC system, the weights can be set based on the high-level policies, such as application priorities, and the proportional-share scheduler would enforce such policies among the applications that share the parallel file system's I/O bandwidth. For example, if two applications are assigned weights 4 and 1, then their shares of the system's I/O bandwidth should be 80% and 20%, respectively, whenever their combined demand exceeds the total available bandwidth. Because only the relative values of the weights matter to the bandwidth allocation, in the paper, the weight assignment to the applications is specified in terms of the ratio among the weights.

SFQ is a classic proportional-share scheduler which is computationally efficient and work conserving with theoretically provable fairness [10]. Because the proposed new scheduler is based on SFQ, a brief summary of the classic scheduler is provided here. In essence, SFQ schedules the backlogged requests from different applications using a priority queue, where each request's priority is positively affected by its application's weight and negatively affected by its cost (often estimated based on the size of the request). The scheduler can dispatch only one outstanding request, and it decides which one to dispatch based on the virtual *start time* and *finish time* of the requests in its queue. A new request's start time is generally assigned based on the same application's previous request's finish time, and its finish time equals to its start time plus its cost. Therefore, when the scheduler dispatches the queued requests according to the increasing order of their start times, each competing application is able to get a fair share of service on the shared resource. When applications are assigned different weights (e.g., based on their priorities), these weights are used to adjust the finish times—the cost of a request reduces proportionally to its application's weight—so that the applications receive service proportionally to their weights.

The scheduler is work-conserving in that when an application does not have enough requests to use up its allocated share, the scheduler does not wait for the application and instead immediately dispatches the next request in the queue with the smallest start time. When this application's next request comes, its start time is set to the current global virtual time—which is always advanced by the scheduler based on the start time of its last dispatched request—instead of the application's previous request's finish time, so the service that it did not use due to previous idleness is forfeited.

SFQ(D) [14] is an extension of SFQ designed for proportional sharing of storage resources which are commonly able to handle multiple outstanding requests concurrently. The level of concurrency that the shared storage resource supports is captured by the parameter $\mathscr{D}$ in SFQ(D). The scheduler follows the original SFQ algorithm when assigning timestamps and dispatching queued requests according to the increasing order of their start times, but it allows up to $\mathscr{D}$ outstanding I/Os to be serviced concurrently by the underlying storage in order to take advantage of the available concurrency of the resource.

The choice of $\mathscr{D}$ has implications on both fairness and resource utilization in a real system. Theoretical bound of fairness comes from the assumption that all applications have backlogged requests in the scheduler. However, it is not the case in reality. On one hand, a larger $\mathscr{D}$ allows more concurrent I/Os and a higher utilization of the storage, but it may hurt fairness because it allows a more aggressive workload to dispatch more I/Os while a less aggressive one to lose more share due to the work-conserving nature of the scheduler. As the dispatched I/Os are out of the control of the scheduler, they may also overload the storage and cause significant delays to the following I/Os from other workloads. On the other hand, a smaller $\mathscr{D}$ gives the scheduler a tighter control on the amount of I/O service that a more aggressive workload can steal from others, and allows the less aggressive workloads to establish backlogged I/Os for the scheduler to choose from when it is ready to dispatch more. It can thus improve fairness among the competing workloads but may lead to underutilization of the storage.

Therefore, it is important to strike a balance between fairness and utilization with an optimal $\mathscr{D}$ under mixed I/O workloads, which is a challenging problem addressed by this paper. The inherent limitation of SFQ(D) is that $\mathscr{D}$ does not capture the impact from the size of each I/O. Every dispatched I/O occupies one out of the $\mathscr{D}$ slots, regardless of the size of the I/O, although a larger I/O generates a higher load on the storage than a smaller one. This limitation becomes pronounced when applying SFQ(D) to an HPC parallel storage system, which has been traditionally oriented to service large, sequential I/Os (e.g., from checkpointing) but is also seeing increasingly more small, random I/Os (e.g., from visualization) [7], [19]. The proposed new scheduler addresses this limitation of SFQ(D) and provides fair scheduling of both large and small parallel I/Os as explained in the rest of this section.

Algorithm 1: Dispatching Algorithm of SFQ(D)+ Scheduler with Backfilling

```
Procedure Dispatch()
    // This function is invoked upon the arrival of a new request or the completion of a dispatched request
    Output: The I/O requests to be dispatched $IOs\_to\_dispatch$
1   foreach request r in the scheduler queue do
        // The queue is sorted in the ascending order of the requests' start times
        // $\mathscr{L}_{sum}$ is the total slot cost of dispatched requests
        // $r.\mathscr{L}$ is the slot cost of $r$
2       if $r.\mathscr{L} + \mathscr{L}_{sum} \leq D$ then
3           $IOs\_to\_dispatch.add(r)$
4           $\mathscr{L}_{sum} \leftarrow \mathscr{L}_{sum} + r.\mathscr{L}$
5       end
6   end
7   return $IOs\_to\_dispatch$
```

B. Variable Cost I/O Depth Allocation

The proposed new SFQ(D)+ scheduler recognizes the different costs of outstanding requests in the underlying storage by allocating different number of *slots* of the total I/O depth to them based on their slot cost $\mathscr{L}$. Once $\mathscr{D}$ is chosen, I/Os of different sizes use different numbers of slots, represented by the values of $\mathscr{L}$, out of the total $\mathscr{D}$ slots that the storage supports, and the total slot costs from all outstanding I/Os should not exceed $\mathscr{D}$. The use of the variable slot costs in the algorithm is described in pseudo code in Algorithm 1. This enhancement to the original SFQ(D) algorithm can effectively protect small, low issue rate workloads and provide better fairness when they are contended by large, intensive workloads. Small I/Os are less affected by large I/Os when they are dispatched together. Low issue rate I/Os also wait less for the outstanding large I/Os to complete. SFQ(D)+ takes control of the contention among the outstanding I/Os in the underlying storage before they are dispatched and get out of the hands of the scheduler. I/Os are differentiated not only by their relative order in the scheduler's queue, per the original SFQ(D) algorithm, but also by their *variable slot cost* $\mathscr{L}$ in the shared storage.

A key question to the application of SFQ(D)+ is how to determine the variable slot cost $\mathscr{L}$ of the requests in terms of their use of the underlying storage's I/O depth. It is addressed

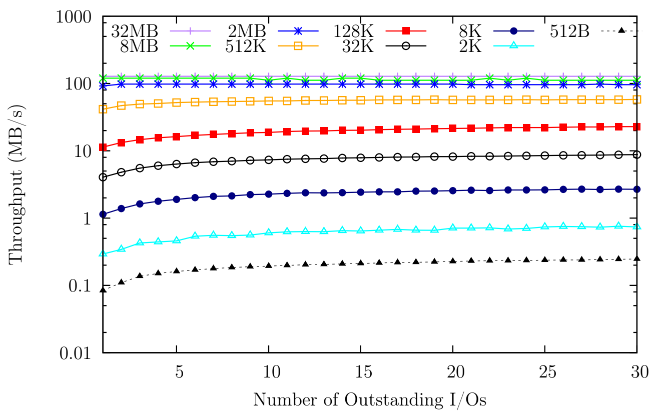


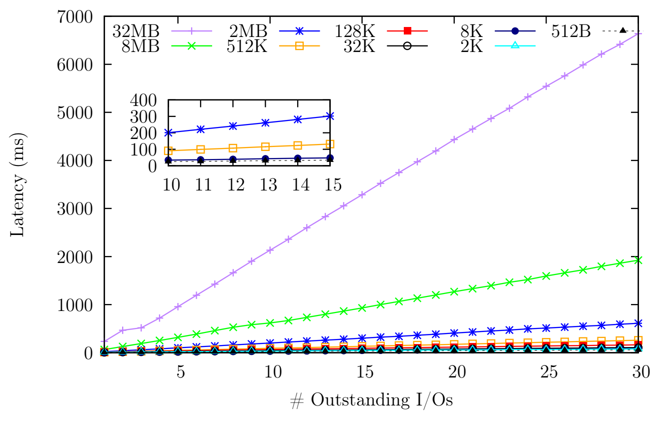
Fig. 2: Throughput models profiled for a data server for different I/O sizes. Large I/Os ((512KB or larger) saturate the storage with less than 5 concurrency I/Os, while small I/Os saturate it with 10 to 15 I/Os.

by profiling the latencies of I/Os of various sizes on the storage. Because I/Os of the same size often require the same amount of processing and incur similar latencies in the storage, they can be considered to have the same slot cost and their latency can be used to estimate the value of $\mathscr{L}$. This methodology is consistent with the common use of I/O size as the estimate of I/O cost for assigning finish tags in classic SFQ. Note that for storage where read and write requests (of the same size) incur different costs (e.g., flash storage), different $\mathscr{L}$ values can be used to further differentiate reads and writes.

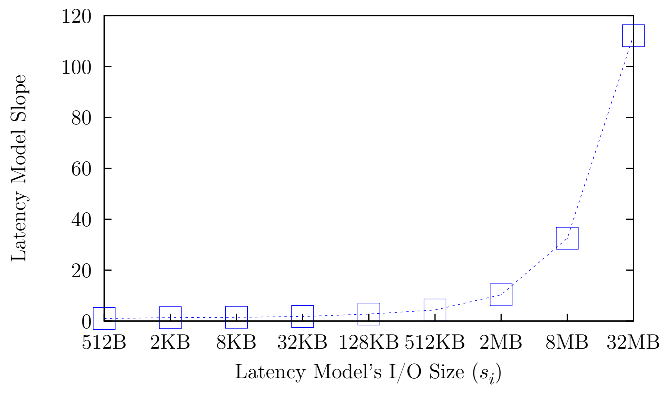
In order to apply this new SFQ(D)+ algorithm to scheduling a shared storage resource, there are two key parameters that need to be determined: first, the $\mathscr{D}$ parameter which represents the number of outstanding I/Os that the storage can handle; and second, the $\mathscr{L}$ parameter which represents the slot cost of an outstanding I/O. Although in principle these two parameters need to be determined for each type of I/Os serviced by the storage, a *single* $\mathscr{D}$ value and a *single* $\mathscr{L}$ value are sufficient to differentiate the large and small I/Os for a parallel file system, which can be determined by profiling the storage as discussed below.

First, the *throughput model* that captures the relationship between the I/O throughput and the number of outstanding I/Os is built to determine the $\mathscr{D}$ parameter of the storage system. The traditional SFQ(D) algorithm [14] also requires profiling to determine $\mathscr{D}$, but it does not consider the dependence of this parameter to the I/O sizes. In SFQ(D)+, the profiling focuses on building a throughput model for small I/Os that the storage serves, so that the cost of large I/Os can be expressed as multiple depth slots where each slot represents the cost of processing a small I/O. In such a throughput model, the throughput generally increases with the growth of the number of outstanding I/Os, and flattens out when the storage becomes saturated. The number of outstanding I/Os that saturates the storage determines the $\mathscr{D}$ parameter for this particular I/O size.

For example, the throughput model of the data server used



(a) Latency models.



(b) The slopes of the latency models from Figure 3a.

Fig. 3: Latency models profiled for a data server for different I/O sizes.

for this paper's experimental evaluation is shown in Figure 2. To confirm that I/Os with different sizes can lead to different values of the $\mathscr{D}$ parameter, the figure shows the throughput model for different I/O sizes, although in practice only the throughput model for small I/Os is required to determine $\mathscr{D}$ for SFQ(D)+. The model was created by issuing random I/O requests directly to the storage while changing the number of outstanding I/Os. The results demonstrate that the storage saturates at different points for different I/O sizes. $\mathscr{D}$ is 1 for I/Os larger than 512KB, and between 10 to 15 for I/Os smaller than 512KB. But as discussed above, SFQ(D)+ requires only the $\mathscr{D}$ for the small I/Os, and we set it to 14 in our experimental evaluation (Section V).

Second, the *latency model* that captures the relationship between the I/O latency and the number of outstanding I/Os is built to determine the $\mathscr{L}$ parameter of the storage system. In general, a large I/O spends the majority of its time on data transfer which grows with the I/O size, whereas a small I/O spends the majority of its time on request processing which is relatively independent of the I/O size. Therefore, by comparing the latency model of different I/O sizes, it is possible to determine the I/O size that differentiates large I/Os from small ones and determine the slot cost of large I/Os with respect to small ones.

Figure 3 shows the latency models for different I/O sizes from the same profiling experiment discussed above. The results confirm that the latency models of small I/Os are similar, whereas the models of large I/Os are quite different. To make this observation clearer, Figure 3b plots the slopes of all the latency models from Figure 3a. It further confirms that the latency slope does not change much for I/O sizes less than 128KB, but it grows proportionally for larger I/O sizes. These latency profiling results show that 128KB is an appropriate value from differentiating small I/Os from large ones, where all I/Os smaller than 128KB can be considered small I/Os and have the same unit slot cost $\mathscr{L} = 1$. Moreover, the results also help determine the slot cost of large I/Os. For example, if the typical large I/Os that the storage serves are 2MB, then the ratio between the latency of 2MB I/Os vs. the latency of small I/Os decides the slot cost of large I/Os, which is in this case 5. It means that when the scheduler dispatches an I/O smaller than 128KB, it uses only one of the $\mathscr{D}$ slots that the underlying storage has; but to dispatch an I/O of 2MB, it has to use 5 out of the $\mathscr{D}$ slots. Note that although small I/Os may come with different sizes, large I/Os are often of fixed size determined by the basic block size that the parallel file system is configured with. For example, in PVFS2 the parallel I/Os to data servers are often issued in 256KB data chunks.

As discussed above, the two key parameters, $\mathscr{D}$ and $\mathscr{L}$, of SFQ(D)+ can be determined using simple profiling experiments which need to be done only once as the throughput and latency models do not change for a given HPC storage stack. The above examples assume that reads and writes have similar I/O costs, but the general approach can also support storage devices such as SSDs that have asymmetric read/write performance by profiling the slot costs $\mathscr{L}$ of reads and writes separately. In this way, SFQ(D)+ is able to understand the actual capacity of the underlying storage for processing different I/Os and the actual costs of these I/Os, in order to achieve fair sharing of the storage for I/Os with diverse sizes.

C. Backfill I/O Dispatching

A side effect of variable I/O slot costs is that the request with the highest priority in the scheduler's queue may not be able to be dispatched immediately even when there are remaining depth slots available but less than the slot cost of the request. The request can be dispatched only when the other outstanding I/Os complete and the number of available slots is restored beyond its slot cost. When there are unoccupied depth slots, the underlying storage is also underutilized. To make efficient use of the storage resources and increase the overall system throughput, SFQ(D)+ employs a new optimization to dispatch I/Os with backfilling so that small I/Os can enter the storage before the large I/Os queued ahead of them when there are idle depth slots available. The use of backfilling in the algorithm is described in Algorithm 1. This optimization matches the work-conserving nature of the SFQ family of schedulers. At the same time, it further promotes the dispatch of small I/Os and is consistent with the SFQ(D)+ algorithm's principle of protecting small I/Os which are vulnerable to the contention from large I/Os.

Backfilling [15] was originally designed to utilize system capacity in a batch job scheduler. As jobs are scheduled based on their order in the queue, the system capacity may not be fully utilized when the remaining capacity is not sufficient to run the job that is currently at the head of the queue. Backfilling allows smaller jobs in the queue that fit the remaining capacity to be scheduled immediately. Akin to the backfilling of jobs, the proposed backfilling for I/O scheduling allows small I/Os to be dispatched before the large ones that are queued ahead of them in order to fully utilize all the slots of the underlying storage's I/O depth.

Specifically, whenever there are free slots available but they do not fit the I/O that is currently at the head of the queue, the SFQ(D)+ scheduler searches its queue for a small, fitting I/O to dispatch instead of waiting for enough free slots to become available for dispatching the large request at the head of the queue. Note that when searching for backfill candidates, the algorithm still follows the start-time order of the requests in the queue and stops when the available free slots are used up or there is no queued request that fits the remaining slots. Therefore, while promoting small I/Os, the scheduler still maintains fairness among small I/Os from different applications. Moreover, to prevent small I/Os from starving the large I/O at the head of the queue, backfilling stops as soon as the total slot cost of the promoted small I/Os equals to or exceeds the cost of the large I/O. In this way, it guarantees that, in the worst case, the large I/O will be dispatched as soon as all the promoted small I/Os complete.

Overall the proposed backfill I/O dispatching complements the variable I/O slot cost in SFQ(D)+ and helps applications with small I/Os mitigate the impact from large, intensive applications in two key aspects. First, when the depth of the underlying storage is fully utilized, only the completion of a large outstanding I/O can warrant the entrance of a similar large I/O, therefore avoiding the dispatch of too many large I/Os to hurt the small I/Os. Second, when large I/Os cannot be dispatched due to the lack of sufficient slots, small I/Os queued by the scheduler can be dispatched out of order. Therefore, the combination of these two techniques in the proposed new SFQ(D)+ scheduler provides the necessary support for achieving fair scheduling in modern, HPC storage systems which face mixed workloads with diverse I/O sizes.

IV. Metadata I/O Scheduling

The above discussion on the SFQ(D)+ scheduler focused on the scheduling of data I/Os on a parallel file system. As the datasets of modern HPC workloads grow, they are also increasingly metadata-intensive [1], [3], [5], [7], [16], [19], [21], which warrants considerations for the metadata I/O performance as well. In fact, it has been observed that the amount of metadata in a system grows at a faster rate than the data. Taking WRF, a real-world scientific application as an example, it entails a variety of intensive metadata operations.

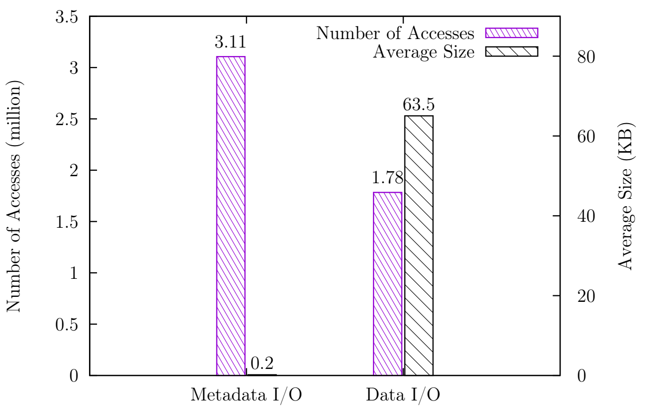


Fig. 4: The comparison between data and metadata I/Os in terms of the number of I/Os (left y-axis) and the average size of the I/Os (right y-axis). The statistics were collected from a WRF run with 64 processes that lasted 10 minutes.

First, for checkpointing, WRF uses a file-per-process, N-to-N access pattern, in which each process writes to its own checkpointing file. The creation of and accesses to a large number of checkpoint files involve substantial metadata operations, which, if not serviced with the adequate throughput, may slow down the checkpoint operations and delay the application progress. Second, for scalable output operations, WRF often employs a group of dedicated I/O processes (in the *I/O quilting* mode) [28] to perform I/Os on a shared output file. These output files involve a large number of concurrent operations to the shared metadata, which can also become a bottleneck if they are not provisioned with fair shares of the metadata service.

As an example, Figure 4 shows the number of metadata I/Os versus data I/Os during a typical run of WRF on PVFS2. A total of 64 MPI processes were spawned to execute WRF on eight nodes, and they used POSIX to access the parallel file system (via the mount points) which uses another eight nodes as both data servers and metadata servers. During the whole run, metadata requests must accompany data requests to fulfill the job, and the total number of metadata requests in fact far exceeds the number of data requests. For example, the metadata I/Os include many GETATTR requests issued by the PVFS2 clients for retrieving the attributes of the input, output, and restart files throughout the run. Each request has an average size of 176 bytes. Meanwhile, most of the data I/Os are 512KB or 64KB in size and split into 64KB or 8KB parallel requests, respectively, across the servers. While this example is specific to PVFS2, as discussed above, parallel file systems in general can have intensive metadata I/Os which are critical to application performance and need to be effectively scheduled for performance management.

Although modern parallel file systems have adopted various techniques to improve the scalability of metadata management, they are not sufficient to guarantee the performance for metadata-intensive applications. For example, parallel file systems can employ distributed metadata servers to share the metadata load, and there is related work on optimizing the load balance across the distributed metadata servers [21]. Some systems start to deploy solid-state storage to hide the latency increase due to the burstiness of metadata accesses [27]. However, such scalability techniques can only improve the overall metadata access performance for applications, and cannot address the performance interference and provide fair sharing of the metadata servers among competing applications. In fact, these parallel file systems do not provide any means to differentiate metadata requests from different applications, and an application's metadata access may be slowed down in unpredictable ways when under contention. Moreover, as an application's data I/Os often depend on its metadata I/Os, the lack of performance isolation on metadata requests can impact data requests and hurt the fairness of sharing the data servers, which further aggravates the performance interference caused by metadata I/O contention.

To solve this problem, vPFS+ also applies the new SFQ(D)+ scheduler to the scheduling of metadata accesses, thereby providing complete support to fair sharing of key parallel storage resources, including both data and metadata servers, and offering comprehensive I/O performance guarantee to HPC applications. Metadata accesses are similar to small data requests in size. But different from data requests, the exact size of the response to a metadata request is not always known before scheduling (e.g., the total number of objects in a directory is unknown to the scheduler until the directory contents are fetched from the server). Nonetheless, the costs of the generally small metadata accesses are similar on the metadata servers (e.g., for the `readdir` request, the contents of a large directory are often sent in multiple smaller responses), and it is much more important to measure metadata throughput in terms of IOPS than MB/s. For example, considering WRF in the above example, metadata servers receive millions of requests in a short period with responses all smaller than 200 bytes. Thus, SFQ(D)+ uses a constant value (10KB) to approximate each metadata I/O's cost, which is large enough to cover most of the metadata I/Os and small enough with respect to 128KB threshold discussed in Section III-B.

Finally, to support distributed metadata servers that modern parallel file systems adopt for scalable metadata management, vPFS+ also extends the virtualization framework that it is built upon to support distributed scheduling of metadata I/Os across the metadata servers and ensure applications' fair-sharing of the system's total metadata service, as how it is done for the distributed scheduling of data I/Os. The amount of service that each individual metadata server provides may be unevenly distributed when applications issue different amounts of requests to their directories and files stored across these servers. For example, PVFS2 [8] stores a directory object (usually at a middle level in a directory tree) and all its underlying metadata objects on a single metadata server. Thus, a frequently accessed directory can make its corresponding metadata server a hotspot and cause uneven distribution of metadata services across the metadata servers.

To achieve total-metadata-service proportional sharing under such scenarios, the vPFS+ schedulers running on the servers coordinate with one another to exchange their local service information, acquire the knowledge of global service distribution, and adjust their scheduling of local metadata requests accordingly. The global synchronization schemes proposed for total-data-service proportional sharing [30] can be adapted to efficiently coordinate the distributed vPFS+ schedulers on metadata scheduling. In particular, in the threshold-driven synchronization scheme, each scheduler synchronizes with the others only when its locally serviced number of metadata operations exceeds a predefined threshold. The worst-case unfairness in the metadata service received by the applications is therefore bounded by this threshold, and by tuning this threshold, the tradeoff between synchronization overhead and service unfairness can be flexibly adjusted based on application and system needs.

V. Evaluation

A. Setup

vPFS+ was prototyped on PVFS2 [8] for the experimental evaluation. PVFS2 is a modern parallel file system implementation, and has comparable performance to other commonly used implementations [24]. It has been often used as the platform for studying various parallel file system problems [1], [3], [7], [21], [32]. The fundamental techniques (virtualization-based parallel I/O scheduling) and algorithms (SFQ(D+) in vPFS+ for managing the performance of parallel file system I/Os are generally applicable to different parallel file system implementations.

This evaluation was done on a test-bed consisting of two clusters, one as compute nodes running a variety of benchmarks and the other as storage nodes running vPFS+ proxies and PVFS2 (version 2.8.2) servers. The storage cluster has eight nodes each with two six-core 2.4GHz AMD Opteron CPUs, 32GB of RAM, and two 500GB 7.2K SAS disks. The compute cluster has eight nodes each with two six-core 2.4GHz Intel Xeon CPUs, 24GB of RAM, and two 1TB 7.2K SAS disks. Both clusters are connected to the same Gigabit Ethernet switch. All the nodes run the Debian 4.3.5-4 Linux with the 2.6.32-5-amd64 kernel and use EXT3 (in the journaling-data mode) as the local file system.

The evaluation considers four types of benchmarks:

Data-intensive parallel I/O benchmark—*IOR* (2.10.3) [17], a typical HPC I/O benchmark, is used to generate parallel I/Os through MPI-IO. IOR issues large sequential reads or writes to represent the I/Os from accessing checkpointing files, which is a major source of I/O traffic in HPC systems. Since there is no computation involved, IOR generates the most intensive I/O workloads.

Metadata-intensive parallel I/O benchmarks—*multi-md-test* from the PVFS2 suite is used to represent applications that are metadata intensive. Multi-md-test measures the performance of concurrent parallel metadata operations from processes that use the POSIX or MPI-IO interface to access the parallel file system. It simulates a burst of back-to-back metadata requests issued to parallel file systems with no computation in between.

Scientific application benchmark—*BTIO* (Block Tri-diagonal solver with I/O subtypes) benchmark from the NASA Parallel Benchmark (NPB) suite (MPI version 3.3.1) [29] is used to represent a typical scientific application with interleaved intensive computation and I/O phases. The problem solving algorithm and its implementation in BTIO make it a good benchmark for parallel I/Os [29]. This paper considers the different I/O access patterns (*Class A* and *Class C*) of BTIO. *Class A* generates 400MB of data and *Class C* generates 6817MB. *Class A* uses *simple* subtype and *Class C* uses *full* subtype of BTIO. The former does not use collective buffering and as a result involves a large number of small I/Os. The latter uses MPI-IO with collective buffering which aggregates and rearranges data on a subset of the participating processes before writing it out.

Real-world scientific application—*WRF* [28] version 3.3, a weather forecast and modeling application widely used for weather research. WRF uses MPI to coordinate parallel computing while using POSIX I/Os for data accesses to the parallel file system mounted on the compute nodes. It is compiled using the *dmpar* configuration (distributed memory option (MPI)), and run with the *em_quarter_ss* test case. This test case produces a simulation of a supercell thunderstorm. The environmental wind makes a “quarter circle” when plotted on a hodograph, and is commonly referred to as “quarter circle shear”. In general, WRF simulates and forecasts weather conditions based on models.

All WRF test cases share the same I/O pattern: WRF sequentially reads input, and sequentially writes output and checkpoint data periodically during the mass calculation for each granular time unit calculated and forecasted. Although the evaluation here focuses on one specific case, it is representative of the common I/O patterns of all WRF cases and the results are thus useful to WRF users in general. In this setup, WRF starts with about 50MB of input data, and for every forecast minute during the storm’s progress, writes to an output file shared by all processes and restart files owned by individual processes. The entire execution generates a total of 70GB of restart files (from 64 processes) and 7GB of final output file. Input, output, and restart files are all stored using NetCDF3 [22], a commonly used scientific data format. The average I/O size is around 32KB on each data server. Note that the choice of data format for WRF will not change the relative comparison between vPFS+ and the native parallel file system, because the management of data and metadata I/O contention is always needed regardless of the data format used.

Based on the profiling results discussed in Section III-B, the parameter $\mathscr{D}$ (the total number of slots that the storage supports) for SFQ(D) and SFQ(D)+ is set to 14, and the parameter $\mathscr{L}_{large}$ (a large I/O’s *slot cost*) for SFQ(D)+ is set

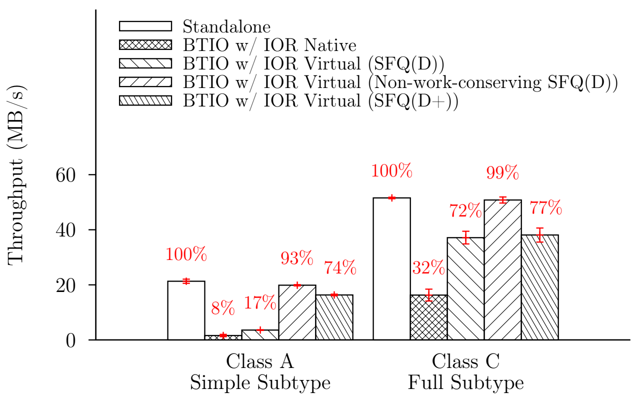


Fig. 5: BTIO's I/O throughput under different configurations. The performance relative to the standalone case is shown on top of the bars.

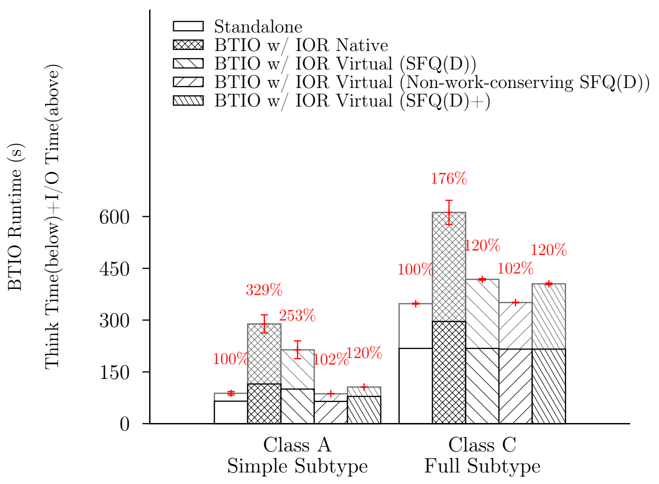


Fig. 6: BTIO's total runtime under different configurations. In each bar, the upper part indicates the amount of I/O Time, and the lower part indicates the amount of Think Time. The performance relative to the standalone case is shown on top of the bars.

to 5, unless otherwise noted[1].

B. BTIO vs. IOR

The first experiment demonstrates how vPFS+ provides performance isolation to applications with small I/Os when under contention from others with intensive, large I/Os on a shared parallel storage system. BTIO is used to model typical HPC applications with interleaved computation and I/Os, and IOR is used to model I/O-intensive HPC workloads (e.g., from checkpointing). BTIO and IOR each have 64 MPI processes running on a separate set of four compute nodes while sharing the eight I/O nodes.

Two types of BTIO workloads are considered in this experiment: *Class C* (writing and reading 6817MB of data) with *full* subtype (using collective buffering) and *Class A* (writing and reading 400MB of data) with *simple* subtype (without collective buffering). A major difference between these two types is that the former issues I/O requests of 4MB to 16MB in size, whereas the latter issues I/Os of 320B in size. Therefore, this experiment can provide a good evaluation of how effectively vPFS+ provides performance isolation to applications with different I/O sizes when under contention from intensive I/Os (continuous 32MB writes from IOR). The goal of the experiment is to minimize the performance impact to BTIO while allowing the competing IOR to fully utilize the unused bandwidth, which is challenging to achieve. The focus is on the performance isolation for BTIO because it is much more vulnerable to the I/O contention due to its much smaller I/Os and lower issue rate, compared to IOR.

Figures 5 and 6 show the I/O throughput and total runtime, respectively, of BTIO under different configurations. When there is no bandwidth management (*BTIO w/ IOR, Native*), BTIO's I/O throughput is reduced by 92.5% in Class A, and 68.4% in Class C, compared to its standalone throughput (*Standalone*). Consequently, BTIO's runtime is increased by 228.8% in Class A, and 75% in Class C. When vPFS+ employs the traditional SFQ(D) scheduler (*BTIO w/ IOR, Virtual (SFQ(D))*), it reduces the slowdown to 153.3% for Class A and 20.3% for Class C, as the throughput is restored to 16.8% and 72.1%, respectively, of the *Standalone* case. As discussed in Section III, the lack of differentiation between large and small I/Os in the traditional SFQ(D) when considering the dispatch of $\mathscr{D}$ requests puts BTIO at an unfair disadvantage. It is also noticeable that Class A's performance is much more challenging to restore than Class C, because of its use of much smaller I/Os which are more sensitive to the interference from IOR's large I/Os.

A non-work-conserving scheduler which strictly throttles an application's bandwidth usage based on its allocation can completely shield BTIO from the impact of I/O contention, but at the cost of poor resource utilization. Specifically, such a scheduler can put the competing application's I/Os temporarily on hold when its completed I/O service exceeds its given bandwidth cap. When running under this non-work-conserving scheduler (*BTIO w/ IOR, Non-work-conserving SFQ(D)*), BTIO can achieve the same level of performance as when it runs alone. Although the non-work-conserving scheduler can provide performance isolation to BTIO, it is not desirable for a shared system because IOR cannot make use of BTIO's spare bandwidth to make progress and the system can be severely underutilized when BTIO's demand is low.

In comparison, the new SFQ(D)+ scheduler is designed to protect small I/Os while still being work-conserving. The result (*BTIO w/ IOR, SFQ(D)+*) in Figure 6 shows that for Class A, SFQ(D)+ can restore BTIO's runtime to 120% of its standalone case, which is 219% better than traditional SFQ(D),

[1]Setting $\mathscr{D}$ to multiples of $\mathscr{L}_{large}$ (e.g., $\mathscr{D}$=10 or 15) will not starve small I/Os in SFQ(D)+. As soon as one large I/O completes, the five slots that it frees up can be used to service backlogged small I/Os

and only 18% worse than the non-work-conserving scheduler. For Class C, SFQ(D)+ cannot achieve more improvement than SFQ(D) because Class C issues I/Os of the same average size as IOR and SFQ(D)+ thus reduces to SFQ(D).

SFQ(D)+ cannot completely restore BTIO's performance as the non-work-conserving scheduler does, because of BTIO's bursty I/Os with low issue rate. Considering one of the 64 BTIO processes, after the initial file creation, the process interleaves four seconds of writes and six seconds of computation for 40 iterations in the first output phase. Then in the verification phase, it repeats three seconds of reads and one second of verification for 40 iterations. In addition, each BTIO process issues only one outstanding I/O. Therefore, BTIO's I/O issue rate is much lower compared to IOR which issues I/Os continuously. Since SFQ(D)+ is work-conserving, spare bandwidth from BTIO has to be yielded to IOR. But when a BTIO process' I/O arrives, it has to wait for the outstanding I/Os of IOR to complete before it can be dispatched. In comparison, the non-work-conserving scheduler would not dispatch any I/O from IOR until BTIO uses up its fair share of the bandwidth.

However, SFQ(D)+ achieves a better balance between resource utilization and performance isolation. When it restores BTIO Class A's performance to 120% of its standalone case, it slows down IOR by only 56% (from 597MB/s to 262MB/s). In comparison, the non-work-conserving scheduler improves BTIO by 20% but has to slow down IOR by up to 99% (from 597MB/s to 0.69MB/s). Overall, SFQ(D)+ achieves 13.81 times better total throughput and resource utilization. Moreover, compared to the traditional SFQ(D), SFQ(D)+ achieves 3.35 times better performance for BTIO with only 10.6% reduction in total throughput. Hence, SFQ(D)+ provides excellent performance isolation for a small I/O application while still making efficient use of the shared storage bandwidth.

C. *WRF vs. IOR*

The second experiment continues to evaluate vPFS+'s support of performance isolation using WRF, a real-world scientific application. WRF reads input at the beginning of each computation iteration and writes checkpointing data (containing dumps of all variables in the model at a certain forecast time) and output data (containing the final forecast variables of the model simulation) periodically. In the experiment, it is run using 64 parallel processes, each issuing reads and writes of sizes between 128KB and 512KB. IOR is used again to create contention using 64 parallel MPI processes each issuing large (8MB) writes. The two applications are run on separate compute nodes but sharing all the eight data nodes. Similarly to the previous experiment, the goal here is to protect the performance of WRF from the I/O contention caused by IOR. But, compared with BTIO, it is even more challenging to provide performance isolation to WRF, because it accesses the parallel file system through the POSIX interface which generates all small I/Os and a large volume of intensive metadata accesses (Figure 4).

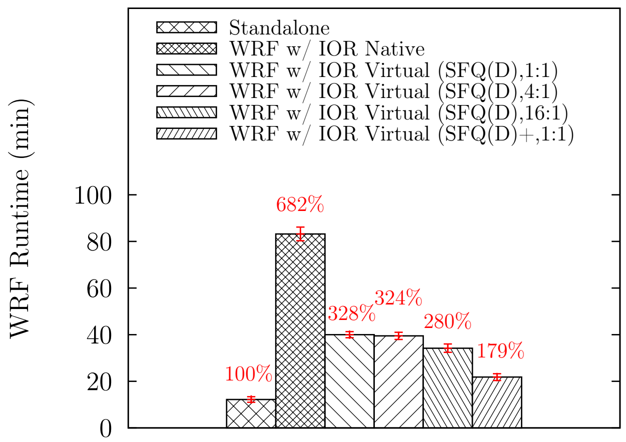


Fig. 7: WRF's runtime under different configurations. The performance relative to the standalone case is shown on top of the bars.

As shown in Figure 7, when contended by IOR, the runtime of WRF is increased by 582% (*WRF w/ IOR Native*). With the SFQ(D) scheduler, vPFS+ can reduce the runtime increase to 180% at best when using a high sharing ratio of 16:1 to favor WRF. In comparison, the new SFQ(D)+ scheduler allows vPFS+ to reduce the slowdown to 79% of the standalone runtime while using a fair 1:1 sharing ratio. This improvement is from the use of variable slot cost to capture large I/Os' actual processing cost in the underlying storage and the use of backfilling to allow small I/Os to be promoted whenever there is free bandwidth available. For WRF, these techniques benefit both its data and metadata I/Os when they are scheduled on the eight PVFS2 nodes which serve as both data and metadata servers. Overall SFQ(D)+ allows WRF to run up to 81% and 281% faster than SFQ(D) and the native case, respectively.

This experiment is also used to evaluate the impact of the $\mathscr{L}$ parameter—the slot cost of large I/Os—in the proposed SFQ(D)+ scheduler. Figure 8 shows the throughput of IOR and WRF as well as their combined throughput with different $\mathscr{L}$ settings. When $\mathscr{L}$ is too small, the scheduler underestimates the cost of large I/Os, and it reduces to the traditional SFQ(D) which considers large I/Os as expensive as small I/Os when making dispatching decisions. The throughput of WRF is thus substantially reduced, whereas IOR does not gain much because it can already saturate the storage. When $\mathscr{L}$ is set too large, the scheduler overestimates the cost of large I/Os, and behaves more like the non-work-conserving scheduler (discussed in Section V-B) where the share allocated to WRF is left idle when WRF cannot fully utilize it. Consequently, the throughput of IOR drops substantially, but WRF cannot gain much due to its low I/O rate, so the combined throughput drops severely too. The experiment shows that adjusting the $\mathscr{L}$ value can balance performance isolation and resource utilization. It also confirms that the choice of $\mathscr{L} = 5$ based on the profiling results is indeed optimal as it provides the best

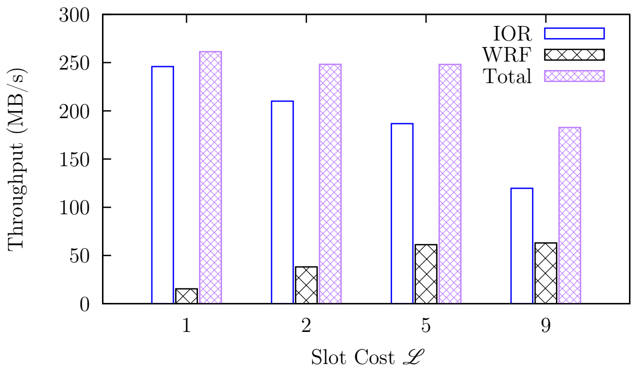


Fig. 8: The throughput of WRF and IOR as well as their combined throughput from SFQ(D)+ with different $\mathscr{L}$ settings for estimating the slot cost of IOR's large I/Os. When $\mathscr{L} = 1$, SFQ(D)+ reduces to SFQ(D). When $\mathscr{L} = 9$, SFQ(D)+ becomes the least work-conserving.

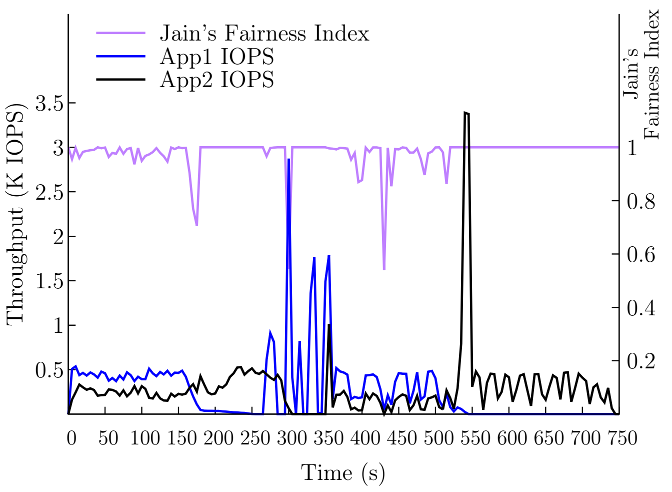


Fig. 9: The metadata I/O throughput of two competing multi-md-test instances (left y-axis) and the Jain's Fairness Index (right y-axis) from a metadata server shared by the two applications.

tradeoff between fairly treating small I/Os vs. large I/Os and fully utilizing the storage bandwidth. Specifically, SFQ(D)+ performs at least 281% better than the non-work-conserving scheduler and 25% better than the traditional SFQ(D) in terms of total system throughput.

D. Metadata Scheduling

The third experiment evaluates vPFS+'s ability of scheduling metadata I/Os and proportionally sharing the parallel file system's metadata service. Multi-md-test from the PVFS2 suite is used to represent a metadata-intensive application. The PVFS2 setup is configured with distributed metadata servers, where each storage node runs both the data server and metadata server. Two instances of the multi-md-test benchmarks (App1 and App2) were used, each with 64 parallel processes, to share the eight data/metadata servers. The assigned weights for App1 and App2 are 2:1. Multi-md-test accesses the mounted parallel file system using the POSIX interface, and the PVFS2 clients on the compute nodes convert the POSIX requests to the PVFS2 requests and send them to the distributed servers. Its execution follows the phases of `mktestdir`, `create`, `write`, `readdir`, `read`, `close`, `rm`, and `rmtestdir`. The `write` and `read` phases involve writing and reading of 400MB data to a file per process.

Figure 9 shows the metadata I/O throughput of the two applications on one of the metadata servers on the left y-axis and Jain's Fairness Index over time on the right y-axis. (The results on the other metadata servers are similar and omitted here.) The throughput data shows two sets of spikes, which represent the high IOPS of the `readdir` phase for both applications, where App1's progress is roughly twice as fast as App2. During the first 160 seconds of the experiment, both applications are performing `mktestdir` and `create` operations, but App1 gets twice the amount of metadata IOPS as App2, which matches their given weights of 2:1. The second overlapped period is from 350 second to 550 second, during which App1 is performing `rm` and `rmtestdir` while App2 is performing `readdir` and `read`. Note that the `read` phase also involves metadata accesses, because a `getattr` request is often issued before a `read` according to the PVFS2 protocol. During this overlapped period, the metadata accesses from the two applications also follow the given 2:1 ratio.

Jain's fairness index [13] is a commonly used metric for evaluating the fairness in resource sharing. It is defined as $\left(\sum_{i=1}^{n}\frac{T_i}{W_i}\right)^2 \Big/ \left(n\sum_{i=1}^{n}\left(\frac{T_i}{W_i}\right)^2\right)$ where T_i and W_i are the throughput and weight, respectively, of Application i in the system. The range of the fairness index is $[0, 1]$ where a larger index value indicates better fairness.

The results in Figure 9 show that vPFS+ achieves excellent fairness (above 0.95 on average in Jain's fairness index) throughout the experiment, despite the fluctuations in total system throughput as the two competing applications execute in different phases dynamically. There are two periods when one of the two applications is in the `write` phase (160s-260s for App1, 300s-360s for App2) and does not issue any metadata request. During these periods, the Fairness Index drops to close to 0.5 only because vPFS+'s work-conserving SFQ(D)+ scheduler allows the application that is more active in metadata I/Os to take away the spare bandwidth from the other and make efficient use of the metadata servers' processing capacity.

E. Overhead

The last experiment studies the performance overhead of vPFS+. Previously reported results [30] have shown that the proxy-based virtualization and scheduling of parallel file system data I/Os has only a small overhead (up to 1% for READ throughput and 3% for WRITE throughput w.r.t. native PVFS2). Therefore, the evaluation here focuses on the overhead of virtualizing and scheduling metadata I/Os in

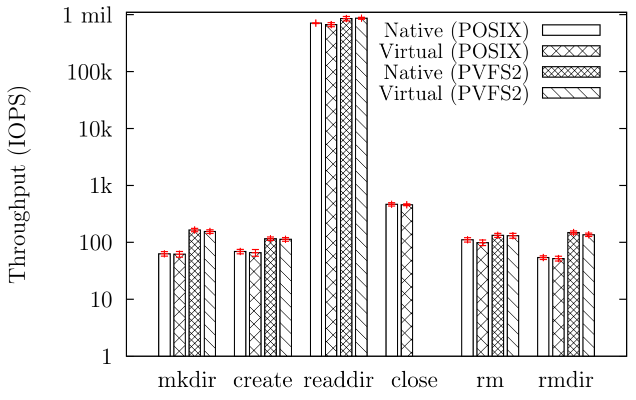


Fig. 10: The overhead of vPFS+ for a metadata-intensive application.

TABLE I: The development cost of vPFS+, estimated by the total number of lines of code (LOC) for the different components of vPFS+

Framework	LOC	Components	LOC
Virtualization	1,695	Interface	694
		TCP	397
		PVFS2	601
Scheduler	3,502	Interface	735
		SFQ(D)	552
		SFQ(D)+	987
		Two-Level	1228
Total	5197		

vPFS+. A set of eight nodes is used to run a total of 128 multi-md-test processes, each issuing 60K metadata operations, and another set of eight nodes is employed as the PVFS2 metadata servers. Figure 10 shows the benchmark's total throughput of metadata operations from vPFS+ (*Virtual*) vs. the native PVFS2 (*Native*). Two different configurations of the benchmark are considered: one issues metadata operations through the POSIX interface (*POSIX*) and the other through the PVFS2 interface (*PVFS2*) directly. The former configuration has a lower throughput because metadata operations have to go through additional layers (Linux virtual file system) to reach the parallel file system. The memory caches on the servers are warmed up before the tests in order to maximize the metadata I/O rate and reveal the worst-case overhead of vPFS+. Overall the overhead is less than 3% when using the PVFS2 API and less than 5% when using POSIX. Note that the PVFS2 API does not provide the `close` call, and the POSIX API does not offer `readdir_stat` and `readdir_plus`, which are omitted in the figure.

F. Cost of Implementation

Finally, vPFS+ is designed as a framework that allows various I/O schedulers to be developed in a modularized fashion and flexibly plugged in for parallel file systems. To evaluate the development effort, Table I summarizes the code complexity of vPFS+. The total number of lines of C code currently in the prototype sums up to $5,197$, including the support for TCP interconnect, PVFS2 parallel file system, and three types of schedulers. To break it down, the virtualization framework requires 1,695 lines of code and the scheduling framework uses 3,502 lines of code. The generic interfaces exposed by these frameworks allow different network transports, parallel file systems, and scheduling algorithms to be incorporated into vPFS+. Specifically, the support for TCP and PVFS2 protocols each costs less than 1,000 lines of code. Different schedulers use from 500 to 1,300 lines of code depending on their complexity. For example, the most complex one is an experimental two-level scheduler [31] that supports both throughput and latency driven I/O scheduling, and it requires 1,228 lines of code.

VI. Related Work

Storage resource management has been studied in related work in order to service competing I/O workloads and meet their desired throughput and latency goals. Such management can be embedded in the shared storage resource's internal scheduler (Cello [25], YFQ [6], PVFS [8]), which has direct control over the resource but requires the internal scheduler to be accessible and modifiable. The management can also be implemented via virtualization by interposing a layer between clients and their shared storage resources (SLEDS [9], SFQ(D) [14], GVFS [33]). This approach does not need any knowledge of the storage resource's internals or any changes to its implementation. vPFS+ follows this approach in order to support existing HPC setups and diverse parallel file systems. Although this virtualization approach has been studied for several storage systems, vPFS+ embodies new designs that address the unique challenges in parallel storage systems for servicing diverse applications.

The majority of storage resource schedulers in the literature focuses on the allocation of a single storage resource (e.g., a storage server, device, or a cluster of interchangeable storage resources) and addresses the local throughput or latency objectives. LexAS [11] was proposed for fair bandwidth scheduling on a storage system with parallel disks, but I/Os are not striped and the scheduling is done with a centralized controller. U-Shape [32] is a related project that tries to achieve application-desired performance by first profiling the application's instantaneous throughput demands and then at runtime scheduling the application's I/Os according to the predicted demands. However, it does not address the often unpredictable contentions in a real-world HPC system where applications with complex behaviors compete on the shared parallel storage system in a convolved manner. In comparison, vPFS+ provides proportional sharing of the data and metadata services for diverse applications without assuming *a priori* knowledge of the applications. DSFQ [26] is a distributed algorithm that can realize total service proportional sharing of distributed storage resources. But it faces challenges in efficient global scheduling when applied to an HPC parallel storage system, which are addressed by vPFS+ and the distributed scheduling framework that it is built upon [30].

vPFS+ leverages the authors' previous work [30] to provide transparent and efficient distributed scheduling of parallel file system I/Os. But as discussed in Section II, it addresses the two major limitations of the previous work in order to support applications with small I/Os and applications that are metadata-intensive. These are non-trivial contributions and are important to today's increasingly diverse HPC environments, as confirmed by a multi-year study of the I/O behaviors from multiple production HPC systems [19].

Recognizing the importance of metadata management, HPC researchers proposed various solutions to improve metadata access performance. OrangeFS [20] and Giga+ [21] studied directory distribution for highly scalable parallel file systems. Arteaga *et al.* proposed parallel file system delegation techniques to offload the management of parallel storage space to applications, relieving the metadata management bottleneck at the parallel file system [3]. Others considered techniques to improve metadata access efficiency in PVFS by precreating objects and batching requests [7], and studied the use of SSD-based buffers for accelerating metadata accesses [27]. These solutions are complementary to vPFS+ in that vPFS+ can benefit from the metadata performance enhancements made by these related solutions while providing the missing control knob for sharing of the metadata service in a fair and efficient manner.

There are related works that also adopt the approach of adding a layer upon an existing parallel file system deployment in order to extend its functionality or improve its performance (pNFS [12], PLFS [4]). These efforts do not address the fair sharing of a parallel file system by concurrent applications and are hence also complementary to this paper's solution.

Finally, the problem of handling diverse I/O workloads has also been studied for other types of systems such as cloud computing systems. MOS [2] is a solution for supporting different types of applications on a cloud object store. vPFS+ complements this related work in that it can be employed by MOS as a low-level I/O scheduler for handling object I/Os of different sizes and further improving its performance isolation and resource efficiency.

VII. Conclusions and Future Work

This paper presents a new solution, vPFS+, to parallel storage management in HPC systems. Today's parallel storage systems are unable to recognize applications' different I/O workloads and unable to satisfy their different I/O performance requirements. This problem is aggravated by the increasing diverse HPC applications with different I/O sizes and intensive data and metadata accesses. vPFS+ addresses this problem based on a parallel file system virtualization framework which allows file system I/Os to be transparently intercepted and differentiated by applications. A new proportional sharing I/O scheduler, SFQ(D)+, is proposed to allow applications with various I/O sizes and issue rates to share the storage with good application-level fairness and system-level utilization. SFQ(D)+ improves over SFQ(D) by taking into account the different I/O depth costs when dispatching small vs. large I/Os, thereby achieving better fairness among diverse workloads. Moreover, the scheduling is extended to support metadata I/Os and to provide performance isolation to metadata-intensive applications. A comprehensive evaluation of vPFS+ using both intensive benchmarks and realistic applications shows that it is able to deliver both high performance and good isolation to diverse HPC applications.

The future work will consider other HPC storage management objectives upon the vPFS+ framework, and extend it beyond proportional bandwidth sharing. In particular, it will consider deadline-driven I/O scheduling to support new HPC applications that are increasingly latency-sensitive. The initial investigation on a two-level I/O scheduler that supports both throughput- and latency-driven parallel I/O scheduling has shown promising results [31]. Moreover, with the emergence of solid-state storage and its adoption in HPC systems, future work will also study the holistic management of SSD and HDD storage resources upon vPFS+.

VIII. Acknowledgement

The authors thank the anonymous reviewers for their helpful comments. This research is sponsored by National Science Foundation CAREER award CNS-1619653 and grants CNS-1562837, CNS-1629888, CMMI-1610282, and IIS-1633381.

References

[1] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. Parallel I/O and the metadata wall. In *Proceedings of the Sixth Workshop on Parallel Data Storage (PDSW)*, pages 13–18, New York, NY, USA, 2011. ACM.

[2] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. MOS: Workload-aware elasticity for cloud object stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 177–188. ACM, 2016.

[3] D. Arteaga and M. Zhao. Towards scalable application checkpointing with parallel file system delegation. In *Proceedings of 6th IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 130–139. IEEE, 2011.

[4] J. Bent, H. Chen, D. Gunter, G. Grider, S. Gutierrez, A. Manzanares, B. McClelland, D. Montoya, J. Nunez, A. Torrez, M. Wingate, G. Gibson, M. Polte, and P. Nowoczinski. PLFS update. High End Computing and File System I/O Workshop, 2010.

[5] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. Storage challenges at Los Alamos National Lab. In *Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, 2012.

[6] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, Washington, DC, USA, 1999. IEEE Computer Society.

[7] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[8] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference (ALS)*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[9] D. Chambliss, G. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. Lee. Performance virtualization for large-scale storage systems. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 109 – 118, Oct. 2003.

[10] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, Oct. 1997.

[11] A. Gulati and P. Varman. Lexicographic QoS scheduling for parallel I/O. In *Proceedings of the seventeenth annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 29–38, New York, NY, USA, 2005. ACM.

[12] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, pages 18–27, Washington, DC, USA, 2005. IEEE Computer Society.

[13] K. R. Jain, W. D.-M. Chiu, and R. W. Hawe. A quantitative measure of fairness and discrimination of resource allocation in shared computer system. In *DEC Research Report TR-301*, 66 Reed Road, Hudson, MA 01749, USA, 1984. Eastern Research Lab, Digital Equipment Corporation.

[14] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the joint International Conference on Measurement and Modeling of Computer systems (SIGMETRICS'04/Performance'04)*, pages 37–48, New York, NY, USA, 2004. ACM.

[15] J. P. Jones and B. Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (IPPS/SPDP'99/JSSPP'99)*, pages 1–16, London, UK, UK, 1999. Springer-Verlag.

[16] S. N. Jones, C. R. Strong, A. Parker-Wood, A. Holloway, and D. D. E. Long. Easing the burdens of HPC file management. In *Proceedings of the Sixth Workshop on Parallel Data Storage (PDSW)*, pages 25–30, New York, NY, USA, 2011. ACM.

[17] R. Klundt. Parallel File System Benchmark. http://sourceforge.net/projects/ior-sio/, 2010.

[18] P. Koutoupis. The Lustre distributed filesystem. *Linux Journal*, 2011(210), Oct. 2011.

[19] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 33–44, New York, NY, USA, 2015. ACM.

[20] M. Moore, D. Bonnie, W. Ligon, N. Mills, S. Yang, B. Ligon, M. Marshall, E. Quarles, S. Sampson, and B. Wilson. OrangeFS: Advancing PVFS. In *Work-in-progress of the 9th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, USA, 2011. USENIX Association.

[21] S. Patil and G. Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST)*, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[22] R. Rew and G. Davis. NetCDF: an interface for scientific data access. *IEEE Computer Graphics and Applications*, 10(4):76–82, July 1990.

[23] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, USA, 2002. USENIX Association.

[24] Z. Sebepou, K. Magoutis, M. Marazakis, and A. Bilas. A comparative experimental study of parallel file systems for large-scale data processing. In *First USENIX Workshop on Large-Scale Computing (LASCO)*, pages 5:1–5:10, Berkeley, CA, USA, 2008. USENIX Association.

[25] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'98/PERFORMANCE'98)*, pages 44–55, New York, NY, USA, 1998. ACM.

[26] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[27] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[28] P. T. Welsh and P. Bogenschutz. Weather research and forecast model: Precipitation prognostics from the WRF model during recent tropical cyclones. In *Proceedings of Interdepartmental Hurricane Conference*, 2005.

[29] P. Wong and R. F. V. der Wijngaart. NAS parallel benchmarks I/O version 2.4. In *NAS Technical Report NAS-03-002*, Moffett Field, CA 94035-1000, USA, 2003. Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center.

[30] Y. Xu, D. Arteaga, M. Zhao, Y. Liu, R. Figueiredo, and S. Seelam. vPFS: Bandwidth virtualization of parallel storage systems. In *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1 –12, april 2012.

[31] Y. Xu and M. Zhao. Two-level throughput and latency IO control for parallel file systems. In *Proceedings of the 8th International Workshop on Feedback Computing*, Berkeley, CA, 2013. USENIX.

[32] X. Zhang, K. Davis, and S. Jiang. QoS support for end users of I/O-intensive applications using shared storage systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 18:1–18:12, New York, NY, USA, 2011. ACM.

[33] M. Zhao, J. Zhang, and R. J. Figueiredo. Distributed file system virtualization techniques supporting on-demand virtual machine environments for grid computing. *Cluster Computing*, 9(1):45–56, Jan. 2006.