vPFS: Bandwidth Virtualization of Parallel Storage Systems

Yiqi Xu, Dulcardo Arteaga, Ming Zhao
Florida International University
{yxu006,darte003,ming}@cs.fiu.edu

Yonggang Liu, Renato Figueiredo
University of Florida
{yongang,renato}@acis.ufl.edu

Seetharami Seelam
IBM T.J. Watson Research Center
sseelam@us.ibm.com

***Abstract*—Existing parallel file systems are unable to differentiate I/Os requests from concurrent applications and meet per-application bandwidth requirements. This limitation prevents applications from meeting their desired Quality of Service (QoS) as high-performance computing (HPC) systems continue to scale up. This paper presents *vPFS*, a new solution to address this challenge through a bandwidth virtualization layer for parallel file systems. vPFS employs user-level parallel file system proxies to interpose requests between native clients and servers and to schedule parallel I/Os from different applications based on configurable bandwidth management policies. vPFS is designed to be generic enough to support various scheduling algorithms and parallel file systems. Its utility and performance are studied with a prototype which virtualizes PVFS2, a widely used parallel file system. Enhanced proportional sharing schedulers are enabled based on the unique characteristics (parallel striped I/Os) and requirement (high throughput) of parallel storage systems. The enhancements include new threshold- and layout-driven scheduling synchronization schemes which reduce global communication overhead while delivering total-service fairness. An experimental evaluation using typical HPC benchmarks (IOR, NPB BTIO) shows that the throughput overhead of vPFS is small (< 3% for write, < 1% for read). It also shows that vPFS can achieve good proportional bandwidth sharing (> 96% of target sharing ratio) for competing applications with diverse I/O patterns.**

Keywords*—*QoS; parallel storage; performance virtualization

I. INTRODUCTION

High-performance computing (HPC) systems are key to solving challenging problems in many science and engineering domains. In these systems, high-performance I/O is achieved through the use of parallel storage systems. Applications in an HPC system share access to the storage infrastructure through a parallel file system based software layer [1][2][3][4]. The I/O bandwidth that an application gets from the storage system determines how fast it can access its data and is critical to its Quality of Service (QoS). In a large HPC system, it is common to have multiple applications running at the same time while sharing and competing for the shared storage. The sharing applications may have distinct I/O characteristics and demands that can result in significant negative impact on their performance.

A limitation of existing parallel storage systems is their inability to recognize the different application I/O workloads — it sees only generic I/O requests arriving from the compute nodes. The storage system is also incapable of satisfying the applications' different I/O bandwidth needs — it is often architected to meet the throughput requirement for the entire system. These limitations prevent applications from efficiently utilizing the HPC resources while achieving their desired QoS. This problem keeps growing with the ever-increasing scale of HPC systems and with the increasing complexity and number of applications running concurrently on these systems. This presents a hurdle for the further scale-up of HPC systems to support many large, data-intensive applications.

This paper presents a new approach, *vPFS*, to address these challenges through the virtualization of existing parallel file systems, achieving application-QoS-driven storage resource management. It is based on 1) the capture of parallel file system I/O requests prior to their dispatch to the storage system, 2) distinguishing and queuing of per-application I/O flows, 3) scheduling of queued I/Os based on application-specific bandwidth allocations, and 4) a proxy-based virtualization design which enables the above parallel I/O interposition and scheduling transparently to existing storage systems and applications. In this way, virtual parallel file systems can be dynamically created upon shared parallel storage resources on a per-application basis, each with a specific allocation of the total available bandwidth.

With vPFS, various I/O scheduling algorithms can be realized at the proxy-based virtualization layer for different storage management objectives. Specifically, this paper considers Start-Time Fair Queueing (SFQ) [5] based algorithms for proportional sharing of storage bandwidth. These algorithms have been applied to different storage systems [6][7], but, to the best of our knowledge, this paper is the first to study their effectiveness for typical HPC parallel storage systems. More importantly, this paper proposes and evaluates improvements to SFQ algorithms motivated by the unique characteristics (parallel striped I/Os) and requirements (high throughput) of parallel storage systems. These enhancements include new threshold-driven and layout-driven synchronization schemes which reduce the global communication overhead while delivering good total-service proportional sharing.

A prototype of vPFS which virtualizes PVFS2 [3], a widely used parallel file system implementation, has been developed and evaluated with experiments using typical parallel computing and I/O benchmarks (IOR [8], NPB BTIO [9]). The results demonstrate that the throughput

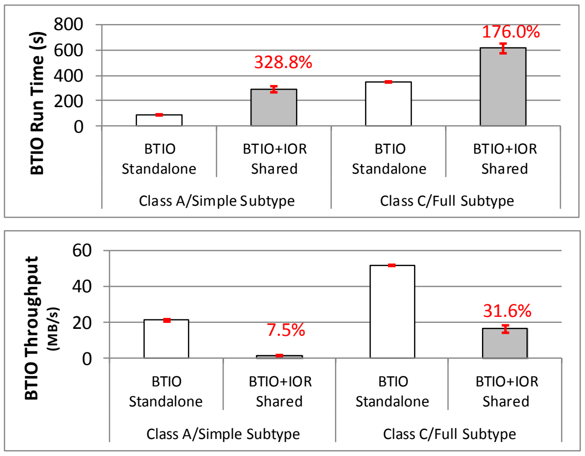


Figure 1. The impact on BTIO's run time and throughput from IOR's I/O contention

overhead from proxy-based virtualization is less than 1% for READ and less than 3% for WRITE (compared to native PVFS2). The results also show that the enhanced SFQ-based schedulers enabled by vPFS achieve good proportional bandwidth sharing (at least 96% of any given target sharing ratio) for competing applications with diverse I/O patterns.

In summary, the contributions of this paper are as follows:

1. A new virtualization-based parallel storage management approach which is, to the best of our knowledge, the first to allow per-application bandwidth allocation in such an environment without modifying existing HPC systems;
2. The design, implementation, and evaluation of vPFS which is demonstrated experimentally to support low-overhead bandwidth management of parallel I/Os;
3. Novel distributed SFQ-based scheduling techniques that fit the architecture of HPC parallel storage and support efficient total-service proportional sharing;
4. The first experimental evaluation of SFQ-based proportional sharing algorithms in parallel file system environments.

The rest of this paper is organized as follows. Section II introduces background and motivation. Section III describes proxy-based virtualization and proportional bandwidth sharing on parallel storage. Section IV presents the evaluation. Section V examines the related work and Section VI concludes this paper.

II. Background and Motivation

A. Limitations of Parallel Storage Management

HPC systems commonly use parallel file systems [1][2][3][4] to manage storage and provide high performance I/O. However, they cannot recognize the diverse demands from different HPC applications which may differ by up to seven orders of magnitude in their I/O performance requirements [10]. For example, WRF [11] requires hundreds of Megabytes of inputs and outputs at the beginning and end of its execution; mpiBLAST [12] needs to load Gigabytes of database only before starting its execution; S3D [13] writes out Gigabytes of restart files periodically in order to tolerate failures during its execution. Moreover, applications running on an HPC system can have different priorities, e.g., due to different levels of urgency or business value, requiring different levels of performance for their I/Os. Because the bandwidth available on a parallel storage system is limited, applications with such distinct I/O needs have to compete for the shared storage bandwidth without any isolation. Hence, per-application allocation of shared parallel storage bandwidth is key to delivering application desired QoS, which is generally lacking in existing HPC systems.

As a motivating example, Figure 1 shows the impact of I/O contention on a typical parallel storage system shared between two applications represented by IOR [8], which issues check-pointing I/Os continuously and NPB BTIO [9], which generates outputs interleaved with computation. Each application has a separate set of compute nodes but they compete for the same set of parallel file system servers. The figure compares the performance of BTIO between when it runs alone (*Standalone*) without any I/O contention, and when it runs concurrently with IOR (*Shared*). The chosen experiment workloads are *Class C* with *full* subtype (using collective I/O [36]) and *Class A* with *simple* subtype (without collective I/O).

The expectation from the BTIO user's perspective when under I/O contention is either no impact (no loss in I/O throughput and no increase in run time) or at least fair sharing (50% loss in I/O throughput and 100% increase in I/O time). However, the results in Figure 1 show that BTIO suffers much more severe performance drop even though the two applications are using separate compute nodes. For Class A with smaller I/Os, the run time is increased by 228.8% as its throughput is reduced by 92.5%. For Class C with larger I/Os, the performance loss is relatively smaller: the total run time is increased by 76% but the throughput is still reduced by 68.4%. The less than 100% increase in total run time is only because of its relatively steady computing time — the I/O time is still increased by 216.5%. (More details on this experiment are presented in Section IV.) These results demonstrate the need for bandwidth management in a parallel storage system, a problem that becomes increasingly pronounced as HPC systems grow in size. It is desirable to provide fair bandwidth sharing so that each application achieves predictable performance regardless of the contention from other applications in a shared HPC system.

A possible solution to this problem is to statically allocate parallel file system servers to each application, but this solution requires significant resources and is not feasible in HPC systems with a large number and dynamic sets of applications. Some systems limit the number of parallel file system clients that an application can access [14][15]. This approach allows an application to always get some bandwidth through its allocated clients. But as shown in the above experiment, it cannot support strong bandwidth isolation because the parallel file system servers are still shared by all the applications without any isolation.

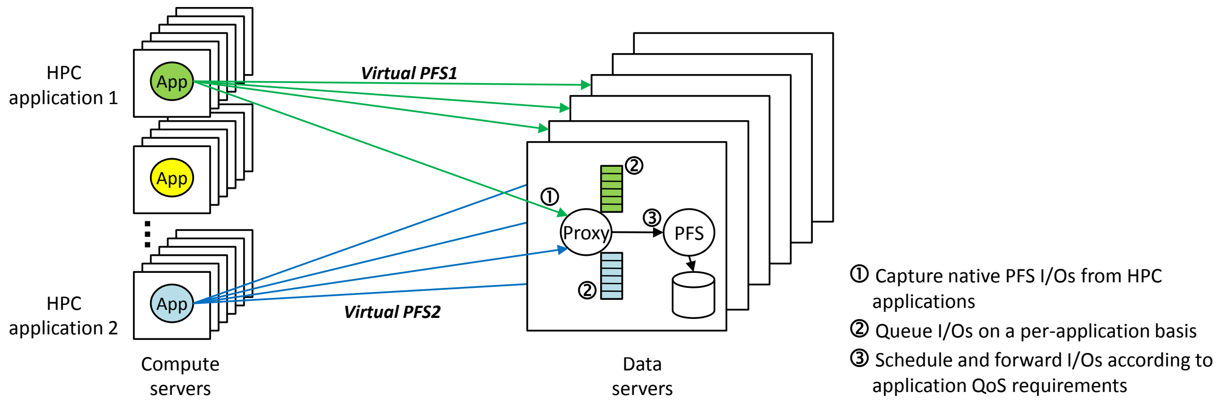

HPC application 1
App
Virtual PFS1
App
App
HPC application 2
Virtual PFS2
Compute servers
①
②
Proxy
③
PFS
②
Data servers
① Capture native PFS I/Os from HPC applications
② Queue I/Os on a per-application basis
③ Schedule and forward I/Os according to application QoS requirements

Figure 2. The architecture of vPFS parallel file system virtualization

B. Limitations of Existing Proportional Bandwidth Sharing Solutions

Proportional-share-based resource allocation is widely used for performance isolation and fair sharing. We focus on approaches based on Start-tag Fair Queueing (SFQ) [5] because of their computational efficiency, work conserving nature, and ability to handle variations in resource capacity. DSFQ [7] is a distributed SFQ algorithm that supports total-service proportional sharing in a distributed storage system. Communication across local schedulers in such a system is necessary for exchanging global scheduling information and determining the delayed service.

However, it is still challenging to providing proportional bandwidth sharing on HPC parallel storage systems using DSFQ. The synchronization required between local schedulers can be prohibitively expensive for large HPC storage systems. DSFQ tries to avoid global synchronization by designing a centralized coordinator to forward (and piggyback cost of) requests between clients and servers in the system [7]. In this way, each server's local scheduler can be aware of the service provided by the others and enforce total-service fairness without explicit synchronization. But such a coordinator must be distributed, in order to be efficient and scalable, assuming that: 1) each coordinator can forward requests destined to any server; 2) each request flow uses all coordinators to forward requests [7]. These assumptions are to ensure that a coordinator can communicate with all local schedulers and always has a uniform chance to piggyback the global scheduling information for each local scheduler. But these assumptions do not hold in typical HPC parallel storage systems.

For high-throughput, a parallel file system client always issues I/O requests directly to the data servers where the corresponding data is stored on (after retrieving the data layout from the metadata server). Therefore, there is nowhere in the parallel storage architecture to place the required coordinators which can receive requests from arbitrary clients, regardless of the data layouts, and forward them to arbitrary data servers. It is possible to modify this architecture to make the data layout opaque to the clients, place the coordinators between the clients and servers, and then force the I/Os to go through the coordinators in a random fashion. However, this design would still be undesirable because 1) it requires a coordinator to forward requests to remote data servers and thus incurs overhead from additional network transfer; 2) it takes away an application's flexibility of specifying data layout, e.g., by specifying layout hints through the MPI-IO interface [16]. These constraints imposed by typical parallel storage architecture motivate the need for a new distributed scheduling design that supports both efficient data access and total-service fairness.

III. vPFS Bandwidth Virtualization

This section presents first the vPFS virtualization which enables per-application bandwidth management on parallel storage, and then the enhanced DSFQ-based schedulers which support efficient total-service proportional sharing upon the virtualization layer.

A. vPFS Virtualization Layer

The virtualization layer in vPFS framework addresses the limitations of existing parallel file systems discussed in Section II.A without changing the interface exposed to applications. It can be integrated with existing deployments transparently, where virtual parallel file systems can be dynamically created on a per-application basis to provide fine-grained bandwidth allocation. The key to such virtualization is to insert a layer of proxy-based parallel I/O indirection between the shared native parallel storage clients and servers, which differentiates per-application I/O flows and enforces their resource allocation. Although this paper focuses only on virtualization-based bandwidth management, such a layer can also enable other new functionalities and behaviors (e.g., I/O remapping for performance and reliability enhancements) — experimental results show that the overhead introduced by a user-level implementation of this layer is small.

In vPFS, a virtual parallel file system is created by spawning a proxy on every parallel file system server that the application needs to access its data (Figure 2). These proxies broker the application's I/Os across the parallel storage, where the requests issued to a data server are first processed and queued by its local proxy and then forwarded to the native parallel file system server for the actual data access. A proxy can be shared by multiple virtual parallel file systems as the proxy instantiates multiple I/O queues to handle the different I/O streams separately. Note that these per-application queues are only conceptual and they can be implemented using a single queue with I/Os tagged using application IDs. Typically HPC systems partition compute nodes across applications so each node, which owns of a unique IP address, executes a single application. Therefore, the proxy can use the compute node's IP to identify I/Os from different applications. However, it might be necessary to further differentiate applications when more than one application runs on a single node in a time-shared or space shared fashion (due to the growing number of cores per node). In such a scenario, each application's I/O can be directed to a specific port of the proxy so that its I/O can be uniquely identified using the port and IP combination.

For high-throughput, vPFS is designed with decentralized proxies which collectively deliver bandwidth virtualization. For total-service fairness of striped I/Os, the virtualization layer recognizes the striping distribution and coordinates the bandwidth allocations across the involved data servers. Therefore, vPFS enables communication and cooperation among distributed proxies to enforce global bandwidth allocation collaboratively.

The placement of vPFS proxies does not have to be restricted to the native parallel file system servers in order to create virtual parallel file systems. They can in fact be placed anywhere along the data path between the clients and servers. For example, the proxies can run on a subset of the compute nodes, which are dedicated to provide I/O services for the rest of the compute nodes. Such proxy placement can be leveraged to deploy vPFS in HPC systems that do not allow any third-party program on the data servers. This placement is also useful when the network bandwidth between the computer and I/O nodes is a bottleneck and needs to be allocated on a per-application basis as well.

The vPFS approach can be applied to support different parallel file system protocols as long as the proxy can understand the protocols and handle I/Os accordingly. Our prototype implementation virtualizes a specific parallel file system, PVFS2; however, we believe that the general vPFS approach is generic enough to support others as well. The user-level virtualization design of vPFS does not require any changes to the existing software and hardware stack. Although the use of proxy for I/O forwarding involves extra I/O processing at the proxy and extra communication with the native data server, our experimental evaluation in Section IV shows that this overhead is small. Alternatively, the performance virtualization can be also implemented directly in the internal scheduler of a parallel file system, but that would require changes to the parallel file system and would be tied to its specific implementation as well.

A variety of parallel I/O scheduling algorithms with different objectives can be deployed at the vPFS virtualization layer by using the proxies to monitor and control I/O executions. In this paper, we focus on proportional sharing and address the challenges of realizing such scheduling efficiently in large-scale parallel storage systems.

The current implementation of vPFS only schedules data requests on a parallel file system and forwards the other meta-data and control requests simply in a FIFO order. However, although not the focus of this paper, vPFS can also achieve performance isolation of meta-data operations, such as the creation of files by proxying and scheduling the relevant meta-data requests.

B. vPFS Distributed Scheduling

In general, to achieve total-service proportional sharing of a parallel file system, a distributed algorithm like DSFQ [7] needs to be enabled upon vPFS. As discussed in Section II.B, the key challenge to implementing such an algorithm is the need of global synchronization across the distributed schedulers. Because each local scheduler sees only the I/Os that it services, it needs to be aware of the service that each competing flow is getting from the other schedulers. In order to efficiently enable total-service proportional sharing, vPFS proposes the following two enhanced DSFQ-based schedulers, which are suited for HPC parallel storage architecture and address the challenge of global synchronization.

1) Threshold-driven Total-Service Proportional Sharing

To enable distributed scheduling, vPFS is enhanced with cooperating proxies. These proxies are responsible of not only scheduling the requests serviced by its local data server but also exchanging the local service information among one another in order to achieve total-service proportional sharing. This distributed scheduling design is suited for typical parallel storage architecture, because, first, it still allows data to directly flow from clients to servers, and second, it does not require a proxy to forward requests to any other remote servers than its local one. However, the need of efficient global scheduling synchronization across the distributed proxies remains to be a challenge.

Global synchronization based on broadcast upon every request is not acceptable in a large system. Instead, the proxies can reduce the frequency of broadcast by batching the costs of a number of locally serviced requests in a single broadcast message. Nonetheless, as a tradeoff, the fairness guarantee offered by the original DSFQ algorithm [7] may be weakened. The overhead of broadcast can be effectively controlled if the proxies synchronize with each other periodically. However, this scheme does not provide a bound on the amount of unsynchronized I/O cost across proxies. As a result, if a server services a large number of requests during a synchronization period, it would cause high fluctuations on the other servers as they try to catch up with the total-service fairness after the synchronization.

In order to achieve efficient total-service proportional sharing with a good unfairness bound, vPFS adopts a *threshold-driven global synchronization* scheme. In this new scheme, a broadcast message is triggered whenever the accumulated cost of arrived requests from a flow f on the local server exceeds a predetermined threshold C_f^{prop}. In this way, the degree of divergence from the perfect total-service fairness is bounded by the threshold C_f^{prop}, because no flow would get unfair extra service more than this bound. This scheme can also limit the fluctuation after each synchronization to the extent of the threshold. Formally, the unfairness bound in this scheme can be described by the theorem:

Theorem 1 Under the threshold-driven total-service proportional sharing algorithm, assume during interval $[t_1, t_2]$, f and g are both continuously backlogged at data server A. The unfairness is defined as the difference between the aggregate costs of requests completed in the entire system for the two flows during the entire interval, normalized by their weights. It is bounded as follows:

$$\left|\frac{W_f(t_1,t_2)}{\emptyset_f} - \frac{W_g(t_1,t_2)}{\emptyset_g}\right| \le \left((D_s + D_d)N_f + 1\right)\frac{C_{f,A}^{max}}{\emptyset_f}$$
$$+\left((D_s + D_d)N_g + 1\right)\frac{C_{g,A}^{max}}{\emptyset_g} + (D_s + 1)\left(\frac{BC_{f,A}^{max}}{\emptyset_f} + \frac{BC_{g,A}^{max}}{\emptyset_g}\right)$$
$$+2(N_f - 1)\frac{C_f^{prop}}{\emptyset_f} + 2(N_g - 1)\frac{C_g^{prop}}{\emptyset_g}$$

In the formula, $W_f(t_1, t_2)$ and $W_g(t_1, t_2)$ are the aggregate costs of requests completed in the entire system for flow *f* and *g*. $\emptyset_f$ and $\emptyset_g$ are the weights of flow *f* and *g*. D_s and D_d are the depths of the scheduler and disks on data server *A*. For any flow *h*, N_h is the number of servers providing service to flow *h*. $C_{h,K}^{max}$ denotes the maximum cost of a single request from flow *h* to data server *K*. $BC_{h,K}^{max}$ represents the maximum total cost of requests from flow *h* to the entire system between the arrivals of any two requests from flow *h* to server *K*.

The unfairness bound in Theorem 1 is similar to that in DSFQ [7], but it has one more component $2(N_f - 1)\frac{C_f^{prop}}{\emptyset_f} + 2(N_g - 1)\frac{C_g^{prop}}{\emptyset_g}$ which is due to the threshold-driven synchronization mechanism. Larger C_f^{prop} and C_g^{prop} save overhead from synchronization, but it also leads to a looser bound of unfairness. By setting $C_f^{prop} = 0$ and $C_g^{prop} = 0$, this extra component disappears, in which case the scheduler synchronizes every time it receives a new request and the algorithm reduces to the original DSFQ algorithm. The proof of this theorem is similar to that in DSFQ [7] so it is omitted for the sake of space. The full proof can be found in our technical report [37].

When implementing this threshold-driven scheme, it can be simplified to use a single threshold, instead of one threshold per flow, on all distributed schedulers. A scheduler issues synchronization when the total cost of requests from all of its serviced flows exceeds this threshold, where the broadcast contains the costs of all of these flows. This simplification will in fact tighten the unfairness bound. Moreover, this single threshold can be conveniently and flexibly adjusted to balance the tradeoff between efficiency and fairness of the total-service proportional sharing.

The synchronization frequency is determined by the threshold and it is a major overhead factor in the broadcast scheme. Therefore a dynamic threshold scheme, which adjusts the threshold value in a continuous manner, is also possible. Because the speed at which the threshold is filled up is a function of I/O size, location and sequentiality, the threshold should be set according to the predicted future access pattern of the flows. This will be part of our future work.

2) *Layout-driven Total-Service Proportional Sharing*

Although the above proposed threshold-driven total-service proportional sharing can substantially reduce the overhead from global scheduling synchronization, the cost will still grow as the number of servers increases in the system. To further reduce the synchronization cost, vPFS also supports a layout-driven scheme in which each distributed local scheduler leverages a flow's file layout information to approximate its global I/O cost from its locally received I/Os. Therefore, frequent global synchronization can be greatly reduced whereas broadcast is required only upon the arrival and departure of flows in the storage system.

A file's layout information, which includes stripe method and parameters, can be either discovered in an I/O request or retrieved from a metadata server. For example, PVFS2 embeds the stripe method name and the specific parameters for this method in every I/O request; if a parallel file system does not do that, such information can still be obtained from the metadata servers. Based on the stripe method used by a flow, a local scheduler can estimate the flow's total service from the striped I/O that it receives locally. For example, in PVFS2, three native stripe methods are implemented: simple stripe, two-dimensional stripe, and variable stripe. When using simple stripe, the total service amount can be approximated by multiplying the request size seen by the local server and the number of servers involved in this flow. When using two-dimensional stripe, each group's I/O size can be constructed by using the factor number (a number indicating how many times to stripe within each group of servers before switching to another group) within each group to approximate the total I/O service. When approximating the total service in variable stripe, different servers' stripe sizes will be used to reconstruct the original I/O request size.

Another necessary parameter for estimating total service is the number of servers involved in each I/O request. The *num_servers* field is embedded together with the stripe information in the PVFS2 I/O requests. In case it is not available in other parallel file system protocols, only one synchronization is required to obtain this information when a new application enters the system. Although it is possible for an application to use different layouts for its files, it is rarely used in practice. For manageability, the application usually prefers using a uniform layout on its entire data set. Therefore, the layout information needs to be retrieved on a per-application basis rather than per-file basis.

By locally calculating total I/O service using the stripe method and parameters as well as the number of involved servers, the global scheduling synchronization can be

TABLE 1. THE OVERHEAD INVOLVED IN THE DEVELOPMENT OF VIRTUALIZATION AND SCHEDULER LAYERS. THE LINES OF CODE ARE COUNTED FOR DIFFERENT PARTS OF VPFS.

Framework	LOC	Component	LOC
Virtualization	1,692	Interface	694
		TCP	397
		PVFS2	601
Scheduler	3,502	Interface	735
		SFQ(D)	552
		DSFQ	987
		Two-Level	1,228
Total	5,194		

virtually eliminated. The reduction in broadcast frequency and message size can lead to substantial saving in processing time and network traffic. Because the synchronization is needed only when an application starts in the HPC system, this cost will be negligible compared with the typical time during which the application stays in the system. Note that the arrival and departure of applications can be estimated by tracking I/O requests or informed by a typical job scheduler (e.g., TORQUE [20], LoadLeveler [21]) commonly used for HPC job management.

Even for applications using mostly small, non-striped I/Os, it is common for the parallel file system to evenly distribute small I/Os on all the involved servers for the sake of performance. Hence, it is still feasible to use the layout information to estimate the total service. In scenarios where this assumption does not hold, vPFS can switch to use the threshold-driven synchronization scheme for more accurate scheduling of such I/Os. In fact, vPFS can make this transition dynamically based on its observation of the I/O patterns.

C. Cost of Implementation

TABLE 1 summarizes the development cost of vPFS. The total lines of code currently in the prototype sums up to 5,194, including the support for TCP interconnect, PVFS2 parallel file system, and three types of schedulers. To break it down, the virtualization framework costs 1,692 lines of code and the scheduling framework costs 3,502 lines of code. The generic interfaces exposed by these frameworks allow different network transports, parallel file systems, and scheduling algorithms to be flexibly incorporated into vPFS. Specifically, the TCP support and PVFS2 interpretation each costs less than 1000 lines of code. Different schedulers cost from 500 to 1,300 lines of code depending on their complexity. A two-level scheduler implementing both bandwidth and latency management [28] is still under development and is estimated to cost more than 1,200 lines of code. The performance overhead of this vPFS prototype is discussed in the next section.

IV. EVALUATION

A. Setup

The PVFS2-based vPFS prototype was implemented on TCP and was evaluated on a test-bed consisting of two clusters, one as compute nodes and the other as I/O nodes running PVFS2 (version 2.8.2) servers. The compute cluster has eight nodes each with two six-core 2.4GHz AMD Opteron CPUs, 32GB of RAM, and one 500GB 7.2K RPM SAS disk, interconnected by a Gigabit Ethernet switch. The server cluster has eight nodes each with two six-core 2.4GHz Intel Xeon CPUs, 24GB of RAM, and one 500GB 7.2K RPM SAS disk. 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, unless otherwise noted) as the local file system.

This evaluation uses IOR (2.10.3) [8], a typical HPC I/O benchmark, to generate parallel I/Os through MPI-IO. IOR can issue large sequential reads or writes to represent the I/Os from accessing check-pointing files, which is a major source of I/O traffic in HPC systems [22]. IOR is also modified to issue random reads and writes and represent other HPC I/O patterns.

The evaluation also uses the BTIO benchmark from the NAS Parallel Benchmark (NPB) suite (MPI version 3.3.1) [9] to represent a typical scientific application with interleaved intensive computation and I/O phases. We consider the *Class A* and *Class C* of BTIO (Class A generates 400MB and Class C generates 6817MB). We also consider both the *full* and *simple* subtypes of BTIO. The former uses MPI I/O with collective buffering which aggregates and rearranges data on a subset of the participating processes before writing it out. The latter does not use collective buffering and as a result involves a large number of small I/Os. All setups are configured to overload the underlying storage system in order to evaluate the effectiveness of performance virtualization under contention.

The different SFQ-based schedulers proposed in Section III.B are evaluated in the experiments. The value of the depth parameter of D used in the algorithms is set to 8, unless otherwise noted. The memory caches on both the clients and servers are cleared before each run. Each experiment is repeated for multiple runs, and both the average and standard deviation values are reported in the following results.

B. Overhead of Proxy-based Virtualization

The first group of experiments studies the performance overhead of vPFS' virtualization and I/O scheduling in terms of throughput, because throughput is the main concern for most HPC applications with large I/O demands. It compares the throughputs between *Native* (native PVFS2 without using proxy), *Virtual* (PVFS2 virtualized with proxy but without any scheduler), and *Virtual-DSFQ* (virtualized PVFS2 with DSFQ-based I/O scheduling). IOR is used in this experiment because it can simulate much more intensive I/O patterns than BTIO. The number of processes used by IOR is 256, evenly distributed on eight physical nodes. To increase the intensity of the request rates and demonstrate the overhead of vPFS under worst-case scenario, in this experiment only, we read the file from a warm cache and use EXT3 in the ordered-data mode. The results in Figure 3 show that the throughput overhead caused by the proxy and its scheduler are small, and when

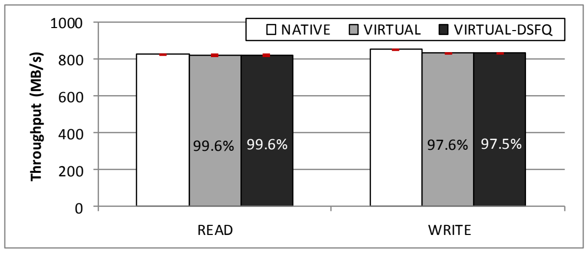


Figure 3. Throughput overhead of vPFS

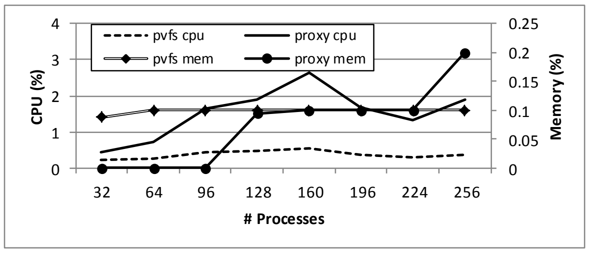


Figure 4. CPU and memory overhead of vPFS

they are combined the total is still less than 1% for READ and 3% for WRITE of the native PVFS2 throughput.

To study the resource overhead of a vPFS proxy, we also observe its CPU and memory usages in an experiment with eight competing applications managed by the threshold-driven enhanced DSFQ scheduler.

Figure 4 shows that the proxy's CPU and memory consumption are low even when it has to handle 256 concurrent I/O flows.

C. *Effectiveness of Proportional Sharing*

The above study confirms that the overhead from proxy-based virtualization and bandwidth management is small and our proposed vPFS approach is feasible. In this subsection, we evaluate the effectiveness of proportional sharing using the enhanced DSFQ-based algorithms on vPFS. Section IV.C.1) uses two IOR instances to model two highly I/O intensive applications. Section IV.C.2) uses eight IOR instances contending for I/O at different times. Section IV.C.3) models a typical scientific application (BTIO) under the impact of other I/O-intensive workload (IOR).

1) *IOR with Various Access Patterns*

This experiment enables threshold-driven DSFQ and shows the ability of vPFS to enforce bandwidth sharing fairness between two applications represented by IOR. The two IOR instances, each forking 128 processes on a separate set of compute nodes, share the data servers in an asymmetric way: *App1* uses only four of the eight data servers whereas *App2* uses all of them. Without any bandwidth management in such an asymmetric setup, the total bandwidth that each application get is proportional to the number of servers it has (1:2). Therefore, this experiment can evaluate whether our proposed distributed schedulers can realize total-service fairness based on any given ratio set between asymmetric *App1* and *App2*. In this experiment, we also consider the proportional sharing between applications with different I/O patterns. Specifically, App1 always issues sequential writes, whereas *App2*'s I/O pattern changes from sequential writes, sequential reads, to random reads and writes (both the offsets and use of read versus write are randomly decided following a uniform distribution). Both applications continuously issue I/O requests so that each flow has a backlog at the server.

Figure 5 shows the proportional sharing achieved between *App1* and *App2* for different read/write combinations and with different total fairness ratios. The achieved ratios are within 1% of the target ratio when it is set to 2:1 and 8:1, and within 3% when it is 32:1. The figure shows that the broadcast scheme can work effectively by synchronizing the total service information and isolating the I/O bandwidth consumed by the different applications. The results shown are obtained using a broadcast threshold of 10MB. It achieves almost the same level of fairness as a much smaller threshold (512KB) which triggers synchronization upon every request. (The details of the 512KB-threshold experiment are omitted here due to the limited space.) Nonetheless, the synchronization overhead from using the 10MB threshold can be reduced by 95% as compared to using the 512KB threshold. These results demonstrate that the distributed scheduler implemented upon vPFS handles the two application's distinct needs with nearly perfect fairness according to any given sharing ratio. Moreover, it does so in an asymmetric setup using efficient global synchronization technique.

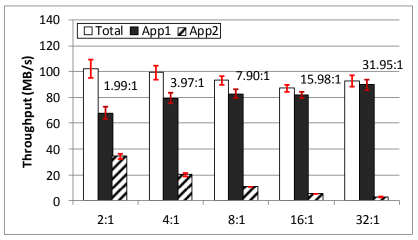
2) *IOR with Dynamic Arrivals*

In this experiment, we evaluate vPFS' ability of handling the dynamic arrivals of more applications. A total of eight applications contend for shared I/O resources. Each application is represented by IOR with 32 processes in sequential writing mode. The odd-numbered applications use four data servers and the even-numbered ones use all of the eight servers. These applications are started one after another and the average arrival interval is 60 seconds. After 420 seconds, all eight applications (256 processes) are present in the system until the end (960th second).

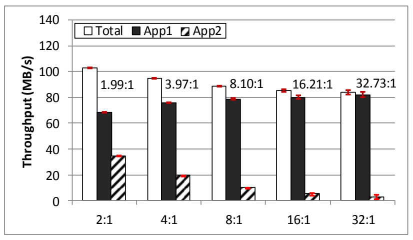
The share of each application is assigned the same value as its ID, i.e., *App1* has a weight of 1 and *App2* has a weight of 2, and so on. Different from last experiment, this experiment employs the layout-driven DSFQ. Figure 6 shows the average and real time unfairness between all the applications every 60 seconds and every 5 seconds throughout this experiment.

We define the unfairness between *n* applications as

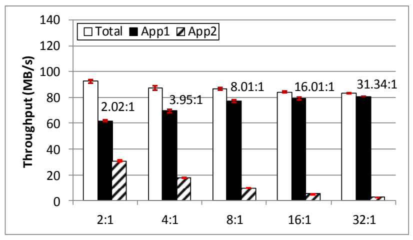
$$\sum_{i=1}^{n} |Throughput_i - Weight_i|$$



(a) Write versus write



(b) Write versus read



(c) Write versus random

Figure 5. Proportional sharing in an asymmetric setup using threshold-driven DSFQ. X-axis is the target share ratio.

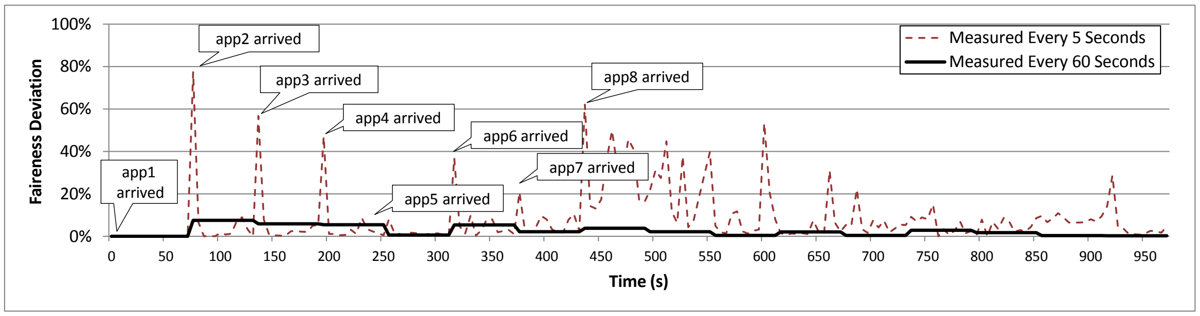


Figure 6. Real time fairness deviation during a 960-second run with 8 dynamically arriving applications

In this formula, $Throughput_i$ and $Weight_i$ are the normalized throughput and normalized weight among the existing applications during each time window. Note that this definition is an extension of that used in Theorem 1 in order to capture unfairness between more than two applications. The value of this formula indicates the sum of each application's fairness deviation (percentage of current available bandwidth unfairly shifted from/to other applications w.r.t. the fair share it should receive). The value ranges from 0% to 200%, the smaller the better, because:

$$0 \le \sum_{i=1}^{n} |Throughput_i - Weight_i|$$
$$\le \sum_{i=1}^{n} |Throughput_i| + |Weight_i|$$
$$= \sum_{i=1}^{n} |Throughput_i| + \sum_{i=1}^{n} |Weight_i| = 2$$

The results in Figure 6 demonstrate that as the number of applications and the number of participating servers change dynamically in the system, the vPFS scheduler is always able to apply layout-driven DSFQ to correctly allocate the total I/O bandwidth to each application according to the specified sharing ratio.

The solid black line in Figure 6, which indicates the average unfairness over every 60 seconds, is always under 9% throughout the experiment. Before the 420th second, the fluctuations of unfairness is relatively larger. This is caused by the arrival of new applications. When a new application enters the system, its requests are not backlogged immediately, so the transient unfairness may increase a lot temporarily. For example, when *App2* enters, it does not have enough number of I/O requests to keep a 2:1 ratio versus *App1* in the scheduler's queue. After the 420th second, the average time unfairness stays well below 4%.

The real time fairness deviation measured every 5 seconds is more interesting to read because it shows the finer grained unfairness. When *App2* enters the system before the 80th second, the spike rising to almost 90% deviation can be explained by the same reason discussed above. This phenomenon recurs when *App3* and *App4* enter, but the impact becomes smaller because the existing applications' established sharing ratio smooth out the impact of new application arrivals.

The pattern of fluctuations of fairness deviations changes after the 420th second for both the 5-second and 60-second measurements. These fluctuations exist because with the eight concurrent applications, the storage system gets overloaded where each parallel file system data servers starts to flush buffered writes in foreground. This explanation is confirmed by observing the CPU's IO wait time using *mpstat* on the data servers, which can be as high as 90% during the flushing. When foreground flushing happens, the I/O processing is temporarily blocked on the servers, which leads to high I/O response times. As such, when we measure the fairness deviation using a 5-second window, we see large spikes when I/Os cannot finish within such a time window. But with a 60-second time window, such spikes are smoothed out in our measurements and it is evident that the fairness deviation is still small overall. Nonetheless, such fluctuations are only an artifact of our intensive experiment which severely overloads the system. They will not show up under the typical HPC storage use.

3) BTIO vs. IOR

In this experiment, we apply vPFS-based storage management to solve the problem discussed in Section II,

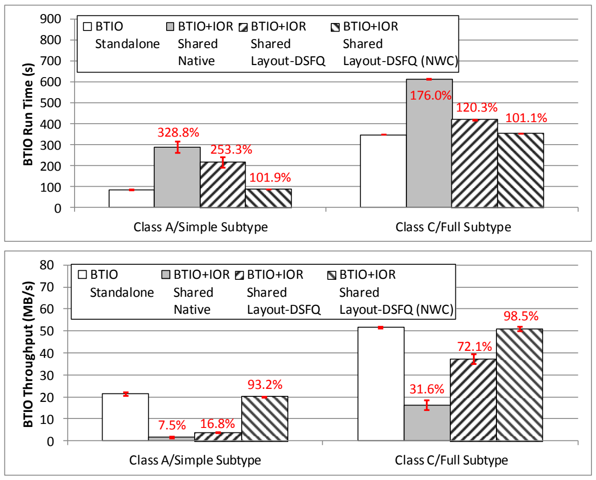


Figure 7. Throughput and run time of BTIO restored by vPFS's different schedulers under contention of IOR

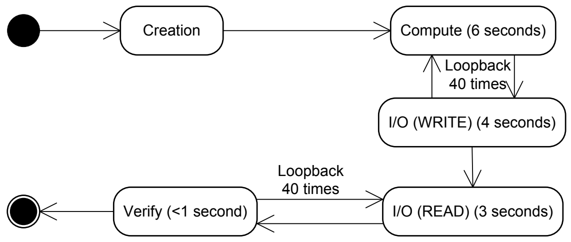


Figure 8. Access pattern of one BTIO process, Class C/Full subtype

i.e., guarantee BTIO's bandwidth allocation and performance under the intensive interference from IOR I/Os. We use the same setup as the one in Section II.A, Figure 1, where BTIO is used to model typical HPC application with interleaved computation and I/O and IOR is used to create intensive, continuous contention on the parallel storage. BTIO and IOR each has 64 MPI processes running on a separate set of compute nodes while sharing the 8 I/O nodes. There are two types of workloads 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 workloads is that the former issues I/O requests of 4 to 16MB in size, whereas the latter issues I/Os of 320B in size. Each IOR process is configured to continuously issue sequential writes of 32MB.

In Figure 7, we show the results from using different vPFS-enabled layout-driven DSFQ schedulers with 32:1 target ratio favoring BTIO. The white bars are the original performance values for different Class/Subtype combinations. The numbers on the other patterned bars indicate the increase in run time or decrease in I/O throughput relative to the original performance.

As discussed in Section II, when there is no bandwidth management (*BTIO+IOR, Shared, Native*), BTIO's run time is increased by 228.8% in Class A and 75% in Class C w.r.t. its standalone run time (*Standalone*). The layout-driven DSFQ (*BTIO+IOR, Shared, Layout-DSFQ*) helps improve the slowdown to 153.3% for class A and 20.3% for Class C, as the BTIO throughput is restored to 16.8% and 72.1%, respectively, of the Standalone case. Notice that Class A's performance is much more challenging to restore than Class C. This is because small I/Os are more susceptible to large I/O contentions from IOR. The static depth used by the DSFQ scheduler, which determines the number of outstanding I/Os, can be also unfair for small I/Os.

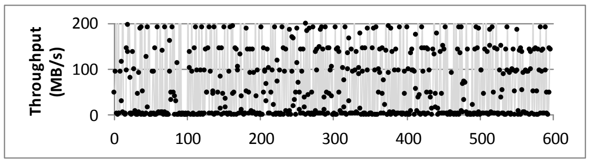
However, even for Class C, the work-conserving DSFQ scheduler cannot fully restore BTIO's performance under the contention from IOR. This is because of BTIO's bursty access pattern. To better illustrate BTIO's I/O pattern and its impact on bandwidth usage, we plot the request patterns of a Class C, full subtype BTIO workload in one of the 64 processes. Figure 8 is a state diagram of the lifecycle of the process. After initial file creation, the process interleaves a 4-second write I/O and a 6-second computation for 40 times in the first output phase. Then the verification phase includes a 3-second read I/O and 1-second verification for 40 loopbacks. In addition, each BTIO process issues only one outstanding I/O. Therefore, the full subtype BTIO workload is quite bursty and has a low issue rate. Since the scheduler used here is work-conserving, spare bandwidth has to be yielded to IOR when BTIO's I/O rate cannot fully utilize the bandwidth share allocated to it. When an I/O burst comes from BTIO, it may also have to wait for the outstanding IOR I/Os to complete.

In order to completely shield the impact of contention, we also implemented a non-work-conserving DSFQ scheduler which strictly throttles an application's bandwidth usage based on its allocation. Specifically, this non-work-conserving scheduler will put an application's I/Os temporarily on hold when its completed I/O service exceeds its given bandwidth cap. When we apply the non-work-conserving (*BTIO+IOR, Shared, Layout-DSFQ (NWC)* in Figure 7) scheduler with a 32:1 ratio, BTIO can achieve the same level of performance as when it runs alone.

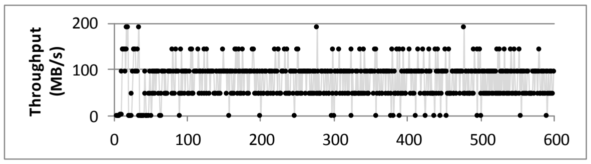
The above discussions demonstrate that there is an interesting tradeoff between resource sharing fairness and resource utilization efficiency. A work-conserving scheduler can fully use available resources, whereas a non-work-conserving one can enforce strict fairness. Nonetheless, vPFS allows such tradeoff to be flexibly balanced based on application and system needs as it enables these various schedulers upon virtualization. It is also conceivable that if we can predict an application's I/O pattern and reserve bandwidth for it in advance of its I/O phases, then it is possible to optimize both fairness and utilization simultaneously. Embedding such intelligence in vPFS I/O scheduling will be considered in our future work.

D. Comparison of Synchronization Schemes

The last group of experiments compares the overhead of global scheduling synchronization between our proposed threshold-driven and layout-driven schemes. In order to magnify the overhead of the two different synchronization schemes, we 1) increase the number of data servers to 96 by using Xen virtual machines (with paravirtualized kernel 2.6.32.5) hosted on the eight server nodes, 2) use the NULL-AIO [23] in PVFS2 to maximize the I/O rate, 3) run

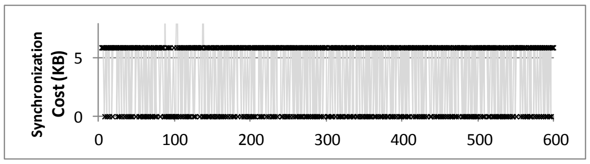


(a) Threshold-driven DSFQ throughput

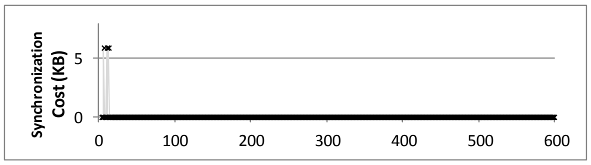


(b) Layout-driven DSFQ throughput

Figure 9. Throughput comparison between threshold-driven DSFQ and layout-driven DSFQ (of one application)



(a) Threshold-driven DSFQ synchronization cost



(b) Layout-driven DSFQ synchronization cost

Figure 10. Synchronization traffic comparison between threshold-driven DSFQ and layout-driven DSFQ (on one server)

TABLE 2. THE COMPARISON OF BANDWIDTH UTILIZATION BETWEEN TWO DIFFERENT SYNCHRONIZATION SCHEMES

Synchronization Scheme	Total Throughput (MB/s)	Standard Deviation (MB/s)
Threshold-driven	540.91	31.59
Layout-driven	612.08	2.21

eight applications, each with 32 parallel processes hosted on a separate physical compute node, 4) set the threshold below the average request size so that every request triggers a broadcast in the threshold-driven scheme, and 5) let all the applications use IOR to generate highly intensive workload. Equal proportional sharing is set as the target for both schemes in the experiment.

The results confirm that both schemes can achieve equally good sharing of the system's total bandwidth (within 3% fairness deviation). More importantly, we monitor the volume of synchronization traffic on each server and the throughput fluctuation of each application to evaluate the overhead of the two schemes and analyze their scalability. Figure 9 shows the throughput of one of the applications and Figure 10 shows the synchronization traffic on one of the servers. (The results obtained from the other applications and servers are similar.) The threshold-driven scheme has a constant need of synchronization and as a result consumes network bandwidth (as well as the CPU cycles for processing the traffic) all the time. Such frequent synchronization also causes high fluctuations on the throughput because every synchronization triggers a re-sorting on the scheduler. In contrast, the layout-driven scheme involves global synchronization only at the entry time of an application. As a result, layout-driven synchronization scheme shows a much smoother throughput in Figure 9 and much less synchronization traffic during the entire experiment in Figure 10. In summary, the layout-based DSFQ scheduler achieves 13.2% higher throughput and 93.0% lower deviation than the threshold-driven DSFQ scheduler (TABLE 2). Meanwhile, it can correctly estimate the total service based on the layout information and thus also achieve tightly bounded proportional sharing within 3% fairness deviation.

V. 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 (e.g., disk schedulers) (Cello [24], Stonehenge [25], YFQ [26], PVFS [27]), 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 (Façade [17], SLEDS [18], SFQ(D) [6], GVFS [19]). This approach does not need any knowledge of the storage resource's internals or any changes to its implementation. It is transparent to the existing storage deployments and supports different types of storage systems.

Although this approach has been proposed for different storage systems, to the best of our knowledge, vPFS is the first to study its application on parallel storage systems. In addition to obtaining the same benefit of transparency as other virtualization-based solutions do, vPFS embodies new designs that address the unique characteristics and requirements of parallel storage systems.

Various scheduling algorithms have been investigated in related storage management solutions. They employ techniques such as virtual clocks, leaky buckets, and credits for proportional sharing [6][25][28], earliest-deadline first (EDF) scheduling to guarantee latency bounds [17], feedback-control with request rate throttling [29], adaptive control of request queue lengths based on latency measurements [30], and scheduling of multi-layer storage resources based on online modeling [31]. The effectiveness of these scheduling algorithms is unknown for a HPC parallel storage system. This gap can be mainly attributed to the fact that there is few existing mechanism that allows effective per-application bandwidth allocation in such a system. Our proposed vPFS approach bridges this gap by enabling various parallel I/O scheduling policies to be instantiated without imposing changes to parallel file system clients and servers. It is also the first to study proportional sharing algorithms on a PFS.

The majority of the 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 [32] 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. DSFQ [7] is a distributed algorithm that can realize total service proportional sharing across all the storage resources that satisfy workload requests. However, as discussed at length in Section II.A, it faces challenges of efficient global scheduling when applied to a HPC parallel storage system, which are addressed by the vPFS and the enhanced algorithms enabled upon it.

U-Shape [40] is a closely related project which tries to achieve application-desired QoS by first profiling the application's instantaneous throughput demands and then dynamically schedule the application's I/Os to meet the predicted demands. In comparison, this paper focuses on proportional bandwidth sharing which can efficiently utilize available parallel storage bandwidth while ensuring total-service fairness among competing applications. In addition, our proposed parallel file system virtualization enables such management without modifying existing HPC storage system software stack.

LACIO [41] also proposed to utilized file layout information on the physical nodes of parallel I/Os, but it is used to optimize the aggregated throughput of the whole system, i.e., to take last level locality into consideration. Ours bears another objective — to infer the global service from locally service I/Os using file layouts, in order to deliver total-service proportional sharing.

Finally, there is related work that also adopts the approach of adding a layer upon an existing parallel file system deployment in order to extend its functionality or optimize its performance (pNFS [33], PLFS [34], ZOID [35]). In addition, cross-server coordination has also been considered in the related work [38][39] to improve spatial locality and performance of I/Os on a parallel file system. These efforts are complementary to this paper's work on virtualization-based proportional sharing of parallel storage bandwidth which, to the best of our knowledge, has not been addressed before.

VI. Conclusion and Future Work

This paper presents a new approach, vPFS, to parallel storage management in HPC systems. Today's parallel storage systems are unable to recognize applications' different I/O workloads and to satisfy their different I/O performance requirements. vPFS addresses this problem through the virtualization of contemporary parallel file systems. Such virtualization allows virtual parallel file systems to be dynamically created upon shared physical storage resources on a per-application basis, where each one gets a specific share of the overall I/O bandwidth. This virtualization layer is implemented via parallel file system proxies which interpose between native clients and servers and capture and forward the native requests according to the scheduling policies.

Upon the vPFS framework, this paper also proposes new distributed schedulers for achieving proportional sharing of a parallel storage system's total bandwidth. These schedulers address the challenges of applying the classic SFQ algorithm [6][7] to parallel storage and enhance it for both high throughput and efficient global scheduling. This paper presents a comprehensive evaluation of the vPFS prototype implemented by virtualizing PVFS2. The results obtained using typical HPC benchmarks, IOR [8] and BTIO [9], show that the vPFS approach is feasible because of its small overhead in terms of throughput and resource usage. Meanwhile, it achieves nearly perfect total-service proportional bandwidth sharing for competing parallel applications with diverse I/O patterns.

In our future work, we will consider other optimization objectives and opportunities upon the vPFS framework, and extend it beyond proportional bandwidth sharing. In particular, we will investigate deadline-driven I/O scheduling to support applications sensitive to latencies. Interactive applications and meta-data intensive applications (with a large number of file creation/deletion operations in a short period of time) are typically latency sensitive and require deadline-driven I/O scheduling. We will study techniques such as pre-fetching and buffering upon vPFS to further improve parallel I/O performance. We also plan to extend the techniques developed in vPFS to the performance virtualization of big data systems such as Hadoop/HDFS.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments. This research is sponsored by the National Science Foundation under grants CCF-0938045 and CCF-0937973 and the Department of Homeland Security under grant 2010-ST-062-000039. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Sun Microsystems, "Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System", White Paper, 2008.

[2] Frank Schmuck and Roger Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", In Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02), Berkeley, CA, USA, Article 19.

[3] PVFS2, Parallel Virtualized File System, URL: http://www.pvfs.org/pvfs2/.

[4] Brent Welch, *et al.*, "Scalable Performance of the Panasas Parallel File System", In Proceddings of the 6th Usenix Conference on File and Storage Technologies (FAST '08).

[5] Pawan Goyal, Harick M. Vin, and Haichen Cheng, "Start Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", IEEE/ACM Trans. Networking, Netw. 5, 5, October 1997, 690-704.

[6] Wei Jin, Jefferey S. Chase, and Jasleen 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). ACM, New York, NY, USA, 37-48.

[7] Yin Wang and Arif Merchant, “Proportional Share Scheduling for Distributed Storage Systems”, In Proceedings of the 5th USENIX conference on File and Storage Technologies (FAST ’07), Berkeley, CA, USA, 4-4.

[8] IOR HPC Benchmark, URL: http://sourceforge.net/projects/ior-sio/.

[9] NAS Parallel Benchmarks, URL: http://www.nas.nasa.gov/publications/npb.html.

[10] Rob Ross, *et al.*, “HPCIWG HPC File Systems and I/O Roadmaps”, High End Computing and File System I/O Workshop, 2007.

[11] Patrick T. Welsh and Peter Bogenschutz, “Weather Research and Forecast Model: Precipitation Prognostics from the WRF Model during Recent Tropical Cyclones”, Interdepartmental Hurricane Conference, 2005, Jacksonville, FL.

[12] Aaron E. Darling, Lucas Carey, and Wu-chun Feng, “The Design, Implementation, and Evaluation of mpiBLAST”, ClusterWorld Conference and Expo, 2003.

[13] Ramanan Sankaran, Evatt R Hawkes, Jacqueline H Chen, Tianfeng Lu and Chung K Law, “Direct Numerical Simulations of Turbulent Lean Premixed Combustion”, Journal of Physics Conference Series, 2006.

[14] Arifa Nisar, Wei-keng Liao, and Alok Choudhary, “Scaling Parallel I/O Performance through I/O Delegate and Caching System”, In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC ’08). IEEE Press, Piscataway, NJ, USA, Article 9, 12 pages.

[15] H. Yu, *et al.*, “High performance file I/O for the Blue Gene/L Supercomputer”, The Twelfth International Symposium on High-Performance Computer Architecture, 2006, pp. 187- 196, 11-15 Feb. 2006.

[16] MPI-IO Library. URL: http://www.mcs.anl.gov/research/projects/mpich2/.

[17] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez, “Façade: Virtual Storage Devices with Performance Guarantees”, In Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST ’03), Berkeley, CA, USA, 131-144.

[18] D.D. Chambliss, *et al.*, “Performance Virtualization for Large-scale Storage Systems”, In Proceedings of the 22nd International Symposium on Reliable Distributed Systems, 2003, pp. 109- 118, 6-18 Oct. 2003.

[19] Ming Zhao, Jian Zhang, and Renato J. Figueiredo, “Distributed File System Virtualization Techniques Supporting On-Demand Virtual Machine Environments for Grid Computing”, Cluster Computing 9, 1 (January 2006), 45-56.

[20] TORQUE Resource Manager, URL: http://www.clusterresources.com/.

[21] IBM LoadLeveler, URL: http://www-03.ibm.com/systems/software/loadleveler/

[22] Fabrizio Petrini and Kei Davis, “Tutorial: Achieving Usability and Efficiency in Large-Scale Parallel Computing Systems”, Euro-Par 2004, Pisa, Italy.

[23] NULL-AIO, URL: http://www.pvfs.org/cvs/pvfs-2-8-branch-docs/doc//pvfs-config-options.php#TroveMethod

[24] Prashant J. Shenoy and Harrick 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), Scott Leutenegger (Ed.), New York, NY, USA, 44-55.

[25] Lan Huang, Gang Peng, and Tzi-cker Chiueh, “Multi-dimensional Storage Virtualization”, In Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’04/Performance ’04), New York, NY, USA, 14-24.

[26] John Bruno, *et al.*, “Disk Scheduling with Quality of Service Guarantees”, In Proceedings of the IEEE International Conference on Multimedia Computing and Systems - Volume 2 (ICMCS ’99), Vol. 2, Washington, DC, USA, 400-.

[27] Robert B. Ross and Walter B. Ligon III, “Server-Side Scheduling in Cluster Parallel I/O Systems”, Calculateurs Parallèles Journal, November, 2001.

[28] Jianyong Zhang, *et al.*, “An Interposed 2-Level I/O Scheduling Framework for Performance Virtualization”, SIGMETRICS Perform. Eval. Rev. 33, 1 (June 2005), 406-407.

[29] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu, “Triage: Performance Differentiation for Storage Systems Using Adaptive Control”, Trans. Storage 1, 4 (November 2005), 457-480.

[30] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger, “PARDA: Proportional Allocation of Resources for Distributed Storage Access”, In Proccedings of the 7th Conference on File and Storage Technologies (FAST ’09), Margo Seltzer and Ric Wheeler (Eds.), Berkeley, CA, USA, 85-98.

[31] Gokul Soundararajan, *et al.*, “Dynamic Resource Allocation for Database Servers Running on Virtual Storage”, In Proccedings of the 7th Conference on File and Storage Technologies (FAST ’09), Margo Seltzer and Ric Wheeler (Eds.), Berkeley, CA, USA, 71-84.

[32] Ajay Gulati and Peter Varman, “Lexicographic QoS Scheduling for Parallel I/O”, In Proceedings of the seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA ’05), New York, NY, USA, 29-38.

[33] Dean Hildebrand and Peter 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 ’05), Washington, DC, USA, 18-27.

[34] John Bent, *et al.*, “PLFS Update”, High End Computing and File System I/O Workshop, 2010.

[35] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman, “ZOID: I/O-forwarding Infrastructure for Petascale Architectures”, In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’08), New York, NY, USA, 153-162.

[36] Rajeev Thakur, William Gropp, and Ewing Lusk, “Data Sieving and Collective I/O in ROMIO”, In Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS ’99), Washington, DC, USA, 182-.

[37] Yiqi Xu, *et al.*, “vPFS: Bandwidth Virtualization of Parallel Storage Systems”, Technical Report, SCIS, FIU, 2011. URL: http://visa.cis.fiu.edu/tiki/tiki-download_file.php?fileId=5.

[38] Huaiming Song, *et al.*, “Server-side I/O Coordination for Parallel File Systems”, In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’11), New York, NY, USA, Article 17, 11 pages.

[39] Xuechen Zhang, Kei Davis, and Song Jiang, “IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination”, In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’10), Washington, DC, USA, 1-11.

[40] Xuechen Zhang, Kei Davis, and Song 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 ’11), New York, NY, USA, Article 18, 12 pages.

[41] Yong Chen, *et al.*, “LACIO: A New Collective I/O Strategy for Parallel I/O Systems”, In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS ’11), Washington, DC, USA, 794-804.