Virtualization-based Bandwidth Management for Parallel Storage Systems

Yiqi Xu, Lixi Wang, Dulcardo Arteaga, Ming Zhao
School of Computing and Information Sciences
Florida International University, Miami, FL
{yxu006,lwang007,darte003,ming}@cis.fiu.edu

Yonggang Liu, Renato Figueiredo
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL
{yonggang,renato}@acis.ufl.edu

Abstract — **This paper presents a new parallel storage management approach which supports the allocation of shared storage bandwidth on a per-application basis. Existing parallel storage systems are unable to differentiate I/Os from different applications and meet per-application bandwidth requirement. This limitation presents a hurdle for applications to achieve their desired performance, which will become even more challenging as high-performance computing (HPC) systems continue to scale up with respect to both the amount of available resources and the number of concurrent applications. This paper proposes a novel solution to address this challenge through the virtualization of parallel file systems (PFSes). Such PFS virtualization is achieved with user-level PFS proxies, which interpose between native PFS clients and servers and schedule the I/Os from different applications according to the resource sharing algorithm (e.g., SFQ(D)). In this way, virtual PFSes can be created on a per-application basis, each with a specific bandwidth share allocated according to its I/O requirement. This approach is applicable to different PFS-based parallel storage systems and can be transparently integrated with existing as well as future HPC systems. A prototype of this approach is implemented upon PVFS2, a widely used PFS, and evaluated with experiments using a typical parallel I/O benchmark (IOR). Results show that this approach's overhead is very small and it achieves effective proportional sharing under different usage scenarios.**

I. Introduction

High-performance computing (HPC) systems are important tools for solving challenging problems in many science and engineering domains. In such systems, parallel storage systems are widely used to provide high-performance storage I/Os to the applications. Applications in an HPC system typically share access to the storage infrastructure through a parallel file system (PFS) based software layer [1][2][3][4]. The I/O bandwidth that an application can get from the parallel storage system is critical to its performance, because it decides how fast they can read and write its data. In a large HPC system, it is common to have multiple applications running at the same time which may have distinct I/O characteristics and requirements. However, an existing parallel storage system is unable to recognize these different application I/O workloads — it only sees 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 architected to meet the throughput requirement for the entire system. These limitations prevent applications from efficiently utilizing the HPC resources while achieving their desired Quality of Service (QoS). This problem will become more severe as HPC systems continue to grow in size and have more applications running concurrently, thereby presenting a hurdle for the further scale-up of HPC systems to support many large, data-intensive applications.

This paper presents a novel approach to addressing the challenges in application QoS driven storage resource management, in order to support the allocation of storage bandwidth on a per-application basis as well as the efficient executions of applications with different I/O demands. Specifically, this new approach provides *per-application storage I/O bandwidth allocation* based on the virtualization of the existing PFSes. Such *PFS virtualization* is realized with user-level proxies that run on the PFS servers, interpose I/Os from compute nodes, and schedule them according to the storage bandwidth allocation. In this way, virtual PFSes are dynamically created upon shared PFS deployment and physical storage resources on a per-application basis, where each virtual PFS gets a specific share of the overall storage bandwidth based on its application's needs. This technique has the potential to enable efficient storage management according to application I/O demand and QoS requirement.

The proposed approach is prototyped upon PVFS2 (Parallel Virtual File System [3]), one of the major PFS implementations. Its performance overhead and effectiveness on bandwidth allocation are evaluated with experiments using a representative parallel I/O benchmark (IOR). The results show that the throughput overhead from proxy-based virtualization is less than 3% compared to the native PVFS2; meanwhile the I/O scheduling (SFQ(D)) implemented upon this virtualization can achieve very good proportional sharing for both symmetric and asymmetric setups.

The rest of this paper is organized as follows. Section II introduces background and related work. Section III gives design and implementation details. Section IV presents an experimental evaluation and Section V concludes this paper.

II. Background And related work

A. Background and Motivation

In a typical HPC system, applications access their data on a parallel storage system which mainly consists of a PFS and its associated storage networks and devices. PFS provides the bridge between the compute and storage infrastructures which are typically connected via a high-speed network. A modern PFS [1][2][3][4] consists of clients, data servers, and metadata servers. *PFS clients* often run on compute nodes and provide the interface to the storage system. *Metadata servers* are responsible for the management of file naming, data location, and file locking. *Data servers* are responsible for performing

This research is sponsored by the National Science Foundation under grants CCF-0938045 and CCF-0937973.

reads and writes on application data, which are usually striped across multiple disks for high throughput.

The parallel storage in an HPC system is often considered as opaque by the applications: it is shared by the compute nodes and serves applications' I/Os in a best-effort manner. As the size of HPC systems grows, it becomes common to have multiple complex parallel and non-parallel applications running concurrently in the same system. Because parallel storage is typically shared by all the compute nodes, it has to service the concurrent I/O requests from all the applications. The storage system is not designed to recognize the different I/Os from applications — it only sees 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 only tuned to satisfy the throughput requirement for the entire system.

However, HPC applications may differ by up to seven orders of magnitude in their I/O performance requirements [7]. For example, WRF [8] requires hundreds of Megabytes of inputs and outputs at the beginning and end of its execution; mpiBLAST [9] needs to load Gigabytes of database only before starting its execution; S3D [10] 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, which also requires different levels of performance for the application I/Os. Since 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. Therefore, per-application storage bandwidth allocation is key to delivering the application's desired QoS, which is generally lacking in existing HPC systems.

Static allocation of PFS data servers to applications can only achieve coarse-grained bandwidth allocation and it is also impractical because large volumes of data cannot be easily swapped in and out as applications do. Some systems limit the number of PFS clients that an application can access [5][6]. This approach ensures that an application always gets certain storage bandwidth, but it cannot support effective bandwidth allocation because the PFS servers are still shared by all the applications without any isolation.

B. *Related Work*

Storage resource management has been studied in the related work in order to service competing I/O workloads and meet their desired throughput and/or latency goals. Such management can be embedded in the shared storage resource's internal scheduler (e.g., disk schedulers) [11][12][13][14], 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 [15][16][17]. Virtualization-based approaches do 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 can support different types of storage systems. The PFS virtualization proposed in this paper is of this type and it can offer these benefits to the resource management of PFS-based storage systems.

Various scheduling algorithms have been investigated in the related storage management solutions. In particular, proportional resource sharing can be realized through fair queuing based algorithms such as Start-Time Fair Queueing (SFQ [18]), which can achieve both work-conserving and strong fairness. SFQ was originally proposed for the scheduling of network resources among competing flows, and it was later extended to support the resource scheduling of a storage system (SFQ(D) [17]). Other algorithms considered for storage resource scheduling include virtual clock based, leaky bucket based, and credit based scheduling for proportional sharing [12][17][19], EDF based scheduling for latency guarantees [15], feedback-control based request rate throttling [20], latency measurements based adaptive control of request queue lengths [21], and online modeling based scheduling of multi-layer storage resources [22].

The majority of the storage resource schedulers in the literature focus on the allocation of a *single storage resource* (e.g., a storage server or device, a cluster of interchangeable storage resources) and address the local throughput or latency objectives. A few related projects also considered global proportional bandwidth sharing for a distributed, parallel storage system [23][24]. In particular, DSFQ [24] is a distributed SFQ algorithm that can realize total service proportional sharing across all the storage resources where the workloads can get service from.

However, the effectiveness of these I/O schedulers is unknown for a PFS-based parallel storage system, which can be mainly attributed to the fact that there is no existing mechanism that allows effective per-application bandwidth allocation in such a system. Our proposed PFS virtualization approach can bridge this gap by enabling such parallel I/O scheduling study and this paper will also present the results from evaluating SFQ(D) on a virtualized PFS deployment.

Finally, there is related work that also adopts the approach of adding a layer upon an existing PFS deployment in order to extend its functionality or optimize its performance [26][27][28]. This paper's work is complementary to these efforts as it focuses on the parallel storage bandwidth allocation which, to the best of our knowledge, has not been addressed before.

III. APPROACH

A. *Design*

Because PFS is the core component of parallel storage management and the main interface between compute and storage infrastructure, this paper proposes to achieve per-application storage resource allocation at the PFS layer. A PFS is designed for general-purpose usage for various types of applications, but it is unable to differentiate their I/Os, which are treated as generic data and metadata requests from the compute nodes. A PFS is deployed statically along with the storage devices, structured and tuned to meet the overall throughput target of the entire system, but it is difficult to customize a PFS configuration according to a specific application's bandwidth needs or to create PFS instances that are tuned on demand for specific application runs.

Our proposed approach addresses these limitations by virtualizing existing PFS deployments so that virtual PFSes can be dynamically created on a per-application basis to provide

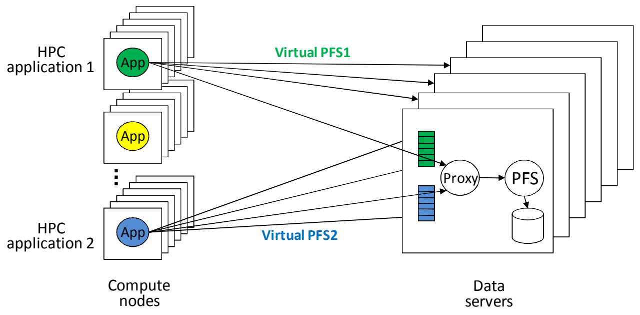

Figure 1. The architecture of PFS virtualization

applications with fine-grained bandwidth allocation. The key to such *PFS virtualization* is to insert a layer of *indirection* upon the *shared physical* software (native PFS) and hardware systems (storage bandwidth) so that *isolated virtual* systems (virtual PFS) can be created with the desired new functionalities and behaviors (per-application bandwidth allocation). The idea behind such virtualization is consistent with other types of virtualization in computer systems such as virtual memory and virtual machines.

Specifically, PFS virtualization can be realized through user-level PFS proxies interposed between native PFS clients and servers (Figure 1). With this technique, a virtual PFS is created by spawning a proxy on every PFS server that the application needs for access to its data. 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 PFS server for the actual data access. A proxy can be shared by multiple virtual PFSes as the proxy instantiates multiple I/O queues to handle the different I/O streams separately. The indirection provided by the PFS proxies forms a layer of virtualization, which allows virtual PFSes to be dynamically created on a per-application basis and to multiplex the physical storage according to the bandwidth allocation in a manner that is transparent to the native PFS. A PFS proxy can enable various scheduling algorithms to achieve different storage resource management objectives, because it can monitor and control the I/Os received from the clients. In particular, proportional sharing scheduling can be implemented upon the proxy-based virtualization layer to allocate bandwidth shares to competing applications.

The key advantages of the proxy-based PFS virtualization technique are that it is transparent to existing PFS deployments in HPC systems and it can be applied to support different PFS protocols *as long as* the proxy can understand the protocols and handle the I/Os accordingly. By virtualizing PFSes with user-level proxies, it can be seamlessly deployed on a wide variety of HPC systems, without any changes to the existing PFS software and hardware stack. However, there is overhead associated with this approach, as the use of proxy for I/O forwarding involves extra I/O processing at the proxy and extra communication with the native PFS data server. Nonetheless, our experimental evaluation (Section IV) shows that this overhead is very small in terms of throughput.

B. Implementation

The proxy-based PFS virtualization is implemented upon PVFS2, one of the widely-used PFS implementations. The proxy's basic functionality is to receive PVFS2 requests from clients and forward them to its local native PVFS2 server; and vice versa, to forward the responses from the native server back to the clients. The proxy performs I/Os asynchronously so that it can handle many requests from different clients concurrently without blocking, thereby avoiding the overhead and complexity of multithreading. To implement a scheduling algorithm, the proxy maintains a separate queue for buffering each application's I/Os and dispatches them based on its algorithm. The proxy only buffers the PVFS2 I/O requests, which do not contain the actual requested data, instead of the actual PVFS2 data flows, so its memory footprint is very small. The proxy provides a generic interface which can be used to implement and plug in different scheduling algorithms. Currently, the proxy has implemented SFQ(D) [17], for basic proportional sharing of its local storage bandwidth.

To virtualize an existing PVFS2 deployment, the proxy needs to be started on each PVFS2 sever with a port number that is different than the native PVFS2 daemon's port number. Correspondingly, the port number specified in the native PVFS2 configuration file (pvfs2tab) on each client needs to be modified so that the PVFS2 requests can be sent to the proxies instead of the native PVFS2 daemons. Each proxy can be dynamically configured via a configuration file, which specifies the clients that belong to each application (IP addresses[*]) and the parameter of the scheduling algorithm (the weight and depth in SFQ(D)).

IV. Evaluation

A. Setup

The proposed PVFS2 virtualization is prototyped on the latest PVFS2 version 2.8.2. The experiment testbed is built with a set of virtual machines (VMs) hosted on a cluster of eight DELL PowerEdge 2970 servers. The cluster is connected by a Gigabit Ethernet and each node has two six-core 2.4GHz Opteron CPUs, 32GB of RAM, and one 500GB 7.2K RPM SAS disk. They run Xen 3.2.1 to support VM environments. Each VM is configured with 1GB RAM and one virtual disk. All physical and virtual machines run para-virtualized 2.6.18.8 kernel in Ubuntu 8.0.4. PVFS2 is set up on these VMs in a way that each physical node has eight client VMs and one server VM. The PVFS2 servers all use EXT3 as the local file system. IOR (2.10.2) [29] is chosen as the benchmark which uses MPI-IO to generate large sequential writes and simulate the typical I/O patterns (such as check-pointing) in HPC applications. Each experiment is repeated for multiple runs and both the average and standard deviation values are reported.

B. Experiment Results

1) Overhead of Proxy-based Virtualization

The first experiment studies the performance overhead of proxy-based virtualization in terms of throughput, since throughput is the main concern for most HPC applications with large I/O requests. Figure 2 compares the throughput between *virtual* PVFS2 (with proxy) and *native* PVFS2 (without proxy) as the number of clients used by IOR increases while the client-to-server ratio is fixed to 16:1. The results show that the

[*] IP-based client identification is feasible because in typical HPC setups the clients are dedicated to different applications based on the compute node partitioning. It is conceivable that such identification can be also implemented by proxying or extending the PFS clients, which will be our future work.

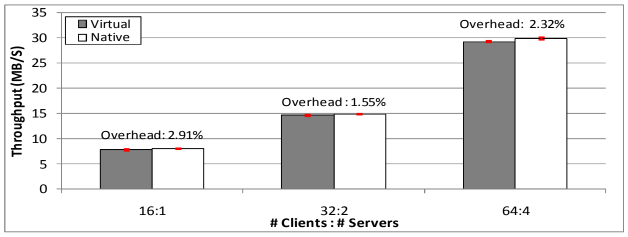

Figure 2. The performance overhead of proxy-based virtualization

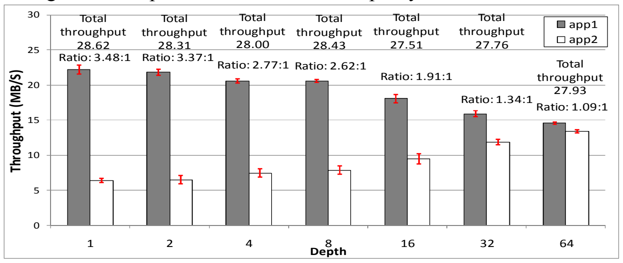

Figure 3. The impact of depth in SFQ(D)

throughput overhead caused by the proxy is less than 3%. During all the experiments discussed in this section, it is also observed that the proxy's resource usage is very small: in average it consumes less than 3% CPU and 1MB RAM. These results prove that the proxy's overhead is negligible for typical HPC applications and the proxy-based virtualization is feasible for parallel storage.

In the following experiments, the proxy-based virtualization is always employed to provide per-application bandwidth allocation. Two IOR instances are used to simulate two concurrent applications, each with a subset of the clients, which compete for the same set of shared servers.

2) *Sensitivity to Depth in SFQ(D)*

The second experiment studies the sensitivity to depth (D) — a parameter in SFQ(D) which specifies the number of outstanding requests that can be serviced by a storage resource concurrently. The choice of depth can have a significant impact on both performance and fairness, because allowing more outstanding requests can increase the resource utilization but hurt the isolation between two competing I/O streams. Figure 3 illustrates the throughputs from two concurrent applications, each with 32 clients, as the depth increases from 1 to 64 (the desired bandwidth ratio is set to 4:1). The results confirm that the use of higher depth causes less resource sharing fairness between the two applications. However, increasing depth does not improve the combined throughput in our experiment, since there is not much margin for improvement in our setup compared to the native PVFS2 throughput. We will further study the impact of depth in scheduling parallel storage on a physical machine based setup in our future work. In the following experiments, the depth is fixed to 2 to support some degree of concurrency in SFQ(D) while keeping the desired share ratio.

3) *Effectiveness of Proportinal Sharing*

The third experiment (Figure 4) studies the effectiveness of proportional bandwidth sharing as the desired share ratio changes from 2:1 to 16:1 between two applications, each using 32 clients in the system. The results show that when the desired ratio is small, the achieved actual ratio is very good (1.9:1 is achieved for the desired 2:1 ratio). However, as the desired ratio increases, the effectiveness of proportional sharing is reduced substantially. When the desired ratio is 16:1, only a 5.8:1 ratio can be actually obtained. We believe that this result is due to relatively low request rate that can be achieved in our experiments. The request rates from competing I/O streams have an important impact to a work-conserving proportional sharing algorithm such as SFQ. If the application with a higher bandwidth allocation cannot issue I/Os fast enough to fill up its queue, it will not be able to fully utilize its allocated share. Meanwhile, the application with a lower allocation can use the bandwidth left idle by the other application and thus bring down the actual share ratio. This conjecture is confirmed by analyzing proxy log files which record queue depths in the experiment.

As a further proof, the fourth experiment intentionally reduces the size of I/O requests from IOR so that the obtainable request rate is increased. In all the other experiments in this section, the request size from IOR is 1MB which is then stripped into four 256KB chunks on the PVFS2 servers. In contrast, in this experiment, the request size is reduced to 64KB. As a result, in Figure 5, the actual achieved ratio is much better than Figure 4. Although it still cannot meet the target ratio when it is 16:1, we believe that it is achievable by using more clients or changing IOR to issue more than one outstanding request. These observations also suggest that the effectiveness of proportional sharing can be improved if the I/O scheduling can be applied at a finer granularity (such as at the PVFS2 data flow level[†]).

The previous two experiments demonstrated that the scheduling algorithm implemented upon the proxy-based virtualization layer can achieve reasonable proportional bandwidth sharing with different ratios, which is impossible to do on the native PVFS2 setup. The next experiment is designed to further prove this point by using an asymmetric setup where the two applications use different numbers of clients. In this setup, the native PVFS2 is not able to achieve equal bandwidth sharing for the applications, because the amount of bandwidth that an application can get is dependent on the number of clients that it uses. Nonetheless, with the scheduling enabled upon the proxy-based virtualization layer, almost perfect fairness can be always obtained even when one application's number of clients is much higher than the other (Figure 6). Notice that the fairness is not as good in the 48:3 case, which is because the application with only three clients cannot generate request fast enough to prevent the other from stealing its allocated bandwidth.

Note that our current SFQ(D)-based scheduling cannot achieve good proportional sharing when the use of PVFS2 servers is asymmetric between the applications, i.e., one application uses more servers than the other, because a proxy can only allocate the bandwidth available on its local server but not globally across the system. Proportional sharing in this scenario requires cooperative scheduling across proxies on different servers and it is part of our ongoing work.

Finally, Figure 7 presents fine-grained throughput measurements collected every five seconds during one run of the third experiment discussed above (with a desired ratio of 2 in Figure 4). These real-time throughput results further demonstrate that good proportional sharing can be also

[†] A PVFS2 I/O request is followed by one or multiple small data flows [3].

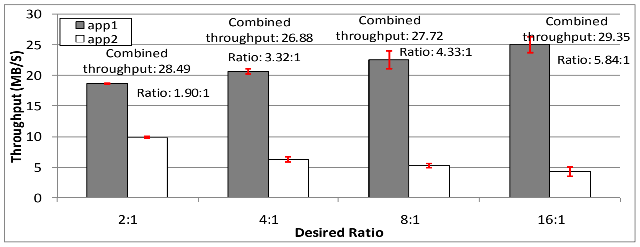

Figure 4. Proportional bandwidth sharing with different ratios

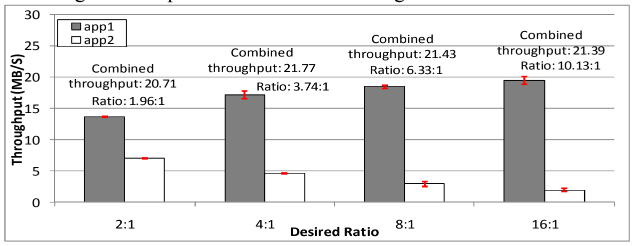

Figure 5. Proportional bandwidth sharing with small I/O requests

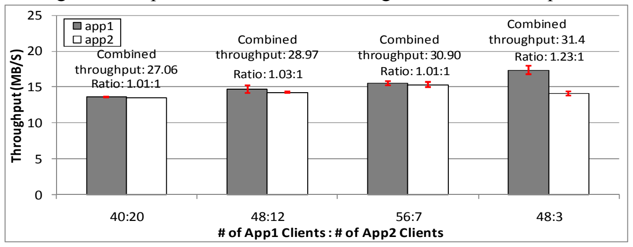

Figure 6. Proportional bandwidth sharing for asymmetric setup

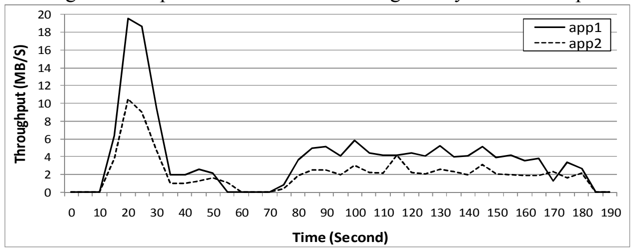

Figure 7. Periodical throughputs of two applications with 2:1 ratio

achieved at such a fine time granularity with only small variations. Notice that there is a large throughput drop in the middle of the run when dirty data buffered in memory starts to be flushed to disk on the server. Nonetheless, a nearly 2:1 ratio is always achieved throughout the entire run.

V. Conclusion and Future Work

This paper presents a new approach 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 needs. This paper proposes a novel technique to address this limitation through the virtualization of contemporary PFSes. Such virtualization allows virtual PFSes to be dynamically created upon shared physical storage resources on a per-application basis, where each virtual PFS gets a specific share of the overall I/O bandwidth according to its application's needs. Our prototype is implemented upon PVFS2 and supports basic proportional sharing based on SFQ(D). The experimental evaluation demonstrates that this approach is feasible because of its negligible throughput and resource usage overhead and it can achieve very effective proportional bandwidth sharing for competing applications. This paper provides a first step towards efficient, application QoS driven storage management. Our future work will further investigate the effectiveness of our approach for applications of different data access patterns and improve it by developing mechanisms for global proportional bandwidth sharing and deadline-driven I/O scheduling.

References

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

[2] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", FAST, pages 231–244, Jan. 2002.

[3] PVFS2. URL: http://www.pvfs.org/pvfs2/.

[4] Brent Welch, et al, "Scalable Performance of the Panasas Parallel File System", FAST, 2008.

[5] A. Nisar, W. Liao, and A. Choudhary, "Scaling Parallel I/O Performance through I/O Delegate and Caching System", SC, 2008.

[6] H. Yu, et al,, "High Performance File I/O for the Blue Gene/L Supercomputer," HPCA, 2006.

[7] Rob Ross, et al., "HPCIWG HPC File Systems and I/O Roadmaps", HPC FSIO 2007 Workshop Report.

[8] P. Welsh, P. Bogenschutz, "Weather Research and Forecast (WRF) Model: Precipitation Prognostics from the WRF Model during Recent Tropical Cyclones", Interdepartmental Hurricane Conference, 2005.

[9] A. Darling, L. Carey, and W. Feng, "The Design, Implementation, and Evaluation of mpiBLAST", ClusterWorld Conf. and Expo, 2003.

[10] R. Sankaran, et al., "Direct Numerical Simulations of Turbulent Lean Premixed Combustion", Journal of Physics Conference Series, 2006.

[11] P. J. Shenoy and H. M. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems", SIGMETRICS, 1998.

[12] L. Huang, G. Peng, and T. cker Chiueh, "Multidimensional Storage Virtualization", SIGMETRICS, 2004.

[13] J. L. Bruno, et al., "Disk Scheduling with Quality of Service Guarantees", in ICMCS, 1999.

[14] R. Ross and W. Ligon, "Server-Side Scheduling in Cluster Parallel I/O Systems", Calculateurs Parallèles Journal, 2001.

[15] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Façade: Virtual Storage Devices with Performance Guarantees", FAST, 2003.

[16] J. Zhang, et al., "Storage Performance Virtualization via Throughput and Latency Control", Trans. Storage, vol. 2, no. 3, 2006.

[17] W. Jin, J. S. Chase, and J. Kaur, "Interposed Proportional Sharing For A Storage Service Utility", SIGMETRICS, 2004.

[18] P. Goyal, H. M. Vin, and H. Cheng, "Start Time Fair Queueing: A Scheduling Algorithm For Integrated Services Packet Switching Networks", IEEE/ACM Trans. Networking, vol. 5, no. 5, 1997.

[19] J. Zhang, et al., "An interposed 2-level I/O scheduling framework for performance virtualization", SIGMETRICS, June 2005.

[20] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance isolation and differentiation for storage systems", IWQoS, 2004.

[21] A. Gulati, et al., "PARDA: Proportional Allocation of Resources for Distributed Storage Access", FAST, 2009.

[22] Gokul Soundararajan, et al., "Dynamic Resource Allocation for Database Servers Running on Virtual Storage", FAST, 2009.

[23] A. Gulati and P. Varman, "Lexicographic QoS Scheduling for Parallel I/O", SPAA, 2005.

[24] Yin Wang and Arif Merchant, "Proportional Share Scheduling for Distributed Storage Systems", FAST, 2007.

[25] H. Kreger, "Web Services Conceptual Architecture", White paper WSCA 1.0, IBM Software Group, 2001.

[26] Dean Hildebrand and Peter Honeyman, "Exporting Storage Systems in a Scalable Manner with pNFS", MSST, 2005.

[27] John Bent, et al, "PLFS Update", HEC FSIO Workshop, 2010.

[28] Kamil Iskra, et al., "ZOID: I/O-forwarding Infrastructure for Petascale Architectures", PPoPP, 2008.

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