Towards Simulation of Parallel File System Scheduling Algorithms with PFSsim

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

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

Abstract

Many high-end computing (HEC) centers and commercial data centers adopt parallel file systems (PFSs) as their storage solutions. As the number of applications concurrently accessing a PFS grows in both quantity and variety, it is expected that scheduling algorithms for data access will play an increasingly important role in PFS service quality. However, it is costly and disruptive to thoroughly research scheduling mechanisms in deployed peta- or exascale systems, compounded by the complexity in scheduling policy implementation and experimental data gathering. While a few parallel file system simulation frameworks have been proposed (e.g., [1,2]), their goals have not been in the scheduling algorithm evaluation. In this paper, we propose PFSsim, a simulator designed for the purpose of evaluating I/O scheduling algorithms in PFS. PFSsim is a trace-driven simulator based on the network simulation framework OMNeT++ [23] and the disk system simulator DiskSim [21]. A flexible scheduler module is provided for scheduling algorithm deployment, and the system characteristics are highly configurable. We have simulated PVFS2 on PFSsim, and the experimental results show that PFSsim is capable of simulating the system characteristics and showing the performance of the scheduling algorithms.

1. Introduction

In recent years, Parallel File Systems (PFSs) such as Lustre [3], PVFS2 [4], Ceph [5], and PanFS [6] have become increasingly popular in high-end computing (HEC) centers and commercial data centers – for instance, as of April 2009, half of the world's top 30 supercomputers use Lustre [7] as their storage solutions. PFSs outperform traditional distributed file systems such as NFS [8] in many application domains. An important reason is that they adopt an object-based storage model [9] and stripe the large-size data into smaller sized objects stored in a distributed manner for high-throughput parallel access and load balancing.

In modern HEC systems and data centers, there are often large numbers of applications which access data with a large variety of Quality-of-Service (QoS) requirements [10]. As such storage systems are predicted to grow in terms of amount of resources and concurrent applications, I/O scheduling strategies that enforce service quality to individual applications are expected to become increasingly important.

There is a considerable amount of work on parallel I/O scheduling, aiming at maximizing overall I/O throughput, such as [11-13]. However, these algorithms are not suitable for many modern systems in that they are not able to provide QoS guarantee to individual applications. Algorithms such as [14-18] address the problem of service isolation in a centralized manner. Nevertheless, many HEC systems face challenges such as intensive data flows from large number of clients and large amount of checkpointing data. In such environments, centralized scheduling algorithms can be limiting from scalability and availability standpoints. There are a few existing decentralized I/O scheduling algorithms which enforce QoS for distributed storage systems, for example, [19,20], but these algorithms need more evaluations in terms of the performance on existing PFSs.

While PFSs are widely adopted in the HEC field, experimental research on corresponding scheduling algorithms is challenging. The two key factors that hamper the testing on real systems are: 1) the cost of scheduler testing on a peta- or exascale file system requires complex deployment and experimental data gathering; 2) experiments with the HEC storage resources can be very disruptive, as these systems typically have high utilization. Under this context, a simulator that allows developers to test and evaluate the PFS scheduling algorithm designs is very valuable. It extricates the developers from complicated deployment headaches in the real systems and cuts their cost in the algorithm development. Even though simulation results are bound to have discrepancies compared to the real system results, the simulation results can offer very useful insights in the performance trends and allow the pruning of the design space before implementation and evaluation on a real testbed or a deployed system.

In this paper, we propose a Parallel File System simulator, PFSsim. Our design objectives for this simulator are: 1) Easy-to-use: scheduling algorithms, PFS characteristics and network topologies can be easily configured at compile-time. 2) Flexible: the simulator should be capable of simulating large variety of scheduling algorithms, and the storage system and networks should be highly customizable. 3) High fidelity: it can accurately model the effect of HEC workloads and scheduling algorithms. 4) Scalable: it should be able to simulate up to thousands of machines for a medium-scale scheduling algorithm study.

The rest of the paper is organized as follows. Section 2 introduces the related work on PFS simulations. Section 3 discusses the abstractive modeling of the PFSs and PFS schedulers. Section 4 illustrates the implementation details of PFSsim. Section 5 shows the simulator validation results. Section 6 concludes this paper and discusses the future work.

2. Related Work

To the best of our knowledge, there are two PFS simulators presented in the literature: one is the IMPIOUS simulator proposed by E. Molina-Estolano, *et. al.* [1], and the other is the simulator developed by P. Carns *et. al.* [2].

The IMPIOUS simulator is developed for fast evaluation of PFS designs. It simulates an abstracted PFS with user-provided file system specifications, which include data placement strategies, replication strategies, locking disciplines and caching strategies. In IMPIOUS, the client modules are configured with the data placement information. They read the I/O traces and directly issue them to the Object Storage Device (OSD) modules according to the data placement specifications. The OSD modules can be simulated with the DiskSim simulator [21] or the "simple disk model"; the former one provides higher accuracy and the latter one has higher efficiency. For the goal of fast and efficient simulation, IMPIOUS simplifies the PFS model by omitting the metadata server modules and related communications, and since the focus is not on the I/O scheduling strategies, it does not support explicit deployment of I/O scheduling algorithms.

The PFS simulator described in [2] focuses on the server-to-server communication mechanisms in PFSs. This simulator is used for testing the overhead of different metadata communication schemes in PVFS. Thus, a detailed TCP/IP based network model is implemented. The authors employed the INET extension [22] of the OMNeT++ discrete event simulation framework [23] to simulate the network. They have implemented a detailed model of PFS and underlying operating system according to PVFS and Linux. This "bottom-up" technique may achieve high fidelity but compromises on the flexibility in simulating other systems. Also, this simulator does not provide a platform for I/O scheduling algorithm deployment.

We take inspiration from these related systems and intend to develop a modularized and customizable system where the emphasis is on the I/O scheduler. Based on this goal, we have developed PFSsim. It adopts the OMNeT++ framework for PFS components and network simulations and uses DiskSim to simulate disk systems. Unlike other simulators, we have implemented the scheduler module, intended to support scheduling algorithm deployment. Since the metadata server and data server daemon components are both implemented in PFSsim, PFSsim modularizes the systems in finer granularity than IMPIOUS. Compared with the simulator proposed by Carns, the customizable modules enable PFSsim to flexibly simulate more systems.

3. System Modeling

To simulate a parallel file system (PFS) for scheduling algorithm testing, we need to construct the system in two phases. First, an authentic PFS simulator need to be built to be able to simulate the performance of a real PFS; second, a scheduler module should be plugged into the PFS simulator, which can accurately show the effects of different scheduling schemes.

In the following subsections, we are going to analyze the mechanisms of real systems by abstracting the PFS deigns and the PFS scheduler designs. These analyses will lay a foundation for the design of our simulator.

3.1. Abstraction of Parallel File Systems

In this subsection, we will first describe the common architecture and mechanisms in PFSs, and then discuss the key factors that contribute to PFS I/O performance.

Considering most of the commonly used PFSs, we find most of them integrate three essential components:

1. There is one or more data servers (also called Object Storage Devices), which are based on the local file systems or the block devices. The application data are stored in the form of fixed-size PFS objects, whose IDs are unique in a global name space.

2. There is one or more metadata servers, which typically manage the mappings from PFS file name space to PFS storage object name space, PFS object placement, as well as the metadata operations.

3. There are a number of PFS clients that run on the system users' machines; they provide the interface (e.g., POSIX) for user applications to access the PFS, and accomplish the transactions with the PFS servers.

For a general PFS, a file data access request (read/write operation) goes through the following steps:

1. Receiving the file I/O request: By calling an API, the system user sends a request {operation, file_path, offset, size} to the PFS client running on the user's machine.

2. Object mapping: The client tries to map the tuple {file_path, offset, size} to a set of objects which hold the requested data. This information is either available locally or require the client to query the metadata servers.

3. Locating the object: The client locates the objects to the data servers storing them. Typically each data server stores the objects with a static range of IDs, and this mapping information is often available on the client.

4. Data transmission: The client sends out data I/O requests to the designated data servers with the information {operation, object_ID}. The data servers reply the requests, and the data I/O starts. The data I/O continues until all the data are transmitted.

Note that for the above process, we have omitted the access permission grant (often conducted on the metadata server) and data locking schemes (conducted on either the metadata server or the data server).

Although different PFSs share the common basic architecture and mechanisms, they differ from each other in many ways, such as data distribution methodology, metadata storage pattern, user API, etc. Nevertheless, there are four aspects that we consider to have significant effects on the I/O performance: metadata management, data placement strategy, data replication model and data caching policy. Therefore, to construct a simulator for various PFSs with fidelity, we should have the above factors considered.

It is known that at least in some cases, metadata operations take a big proportion of file system workloads [24], and also because of lying in the critical path, the metadata management can be very important to the overall I/O performance. Different PFSs use different techniques to manage metadata to achieve different levels of metadata consistency, reliability and access efficiency. For example, Ceph [5] adopts the *dynamic subtree partitioning* technique [25] to distribute the metadata onto multiple metadata servers for high metadata locality and load balance. Lustre [3] enhances metadata reliability by deploying two metadata servers, which includes one "active" server and one "standby" server for failover. In PVFS2 [4], metadata are distributed onto data servers to prevent single point of failure and the performance bottleneck.

Data placement strategies are designed with the basic goal of achieving high I/O parallelism and server utilization/load balancing. But different PFS still vary with each other significantly, for the reason of different usage contexts. Ceph is aiming at providing high reliability and scalability to the large-scale data storage. So it uses the CRUSH (Controlled Replication Under Scalable Hashing) technique [26] to achieve pseudo-random data distribution, which avoids imbalance or load asymmetries. Also, this scheme facilitates the metadata management, since it avoids metadata traffic during data location lookup and reduces the update frequency of the system map. In contrast, aiming to serve the users with higher trust and skills, PVFS2 provides flexible data placement options to the users — it even delegates the users the ability to store data on user-specified data servers.

Data replication and failover models also affect the I/O performance, because for systems with data replication setup, data are written to multiple locations, which may prolong the writing process. For example, with data replication enabled in Ceph, every write operation is committed to both the *primary* OSD and the *replica* OSDs inside a *placement group*. Though Ceph maintains parallelism when forwarding the data to the *replica* OSDs, the costs of data forwarding and synchronization are still non-negligible. Lustre and PVFS2 do not implement explicit data replication models assuming that the replication is managed by the disk systems or the system users.

Data caching on the server side or the client side may improve the PFS I/O performance. But the coherency of the cached data also needs to be managed. PanFS [6] data servers implement write-data caching that aggregates multiple writes for efficient data transmission and better data layout at the OSDs, which may increase the disk I/O rate. Ceph implements the *O_LAZY* flag for *open* operations to relax the coherency requirements for a shared-write file. This facilitates the HPC applications which often have concurrent accesses to different parts of the files. Some PFSs do not implement client caching in their default setup, such as PVFS2. Note that the systems that the PFS data servers running on may also do data caching,

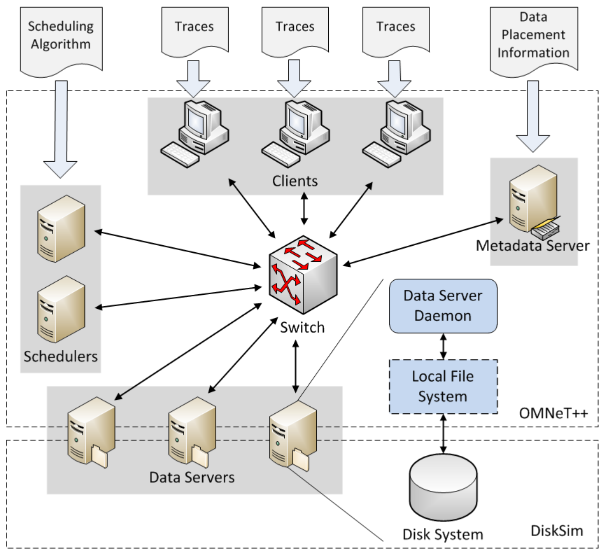


Figure 1. The simulated architecture of an example PFS. The two dash-line frames mean the components inside are simulated by OMNeT++ or DiskSim.

for example, the local file systems on the data server machines.

We have considered the above factors in PFSsim design. For metadata management, by tuning the metadata server module and the network topology, users are able to set up specific metadata storage and access patterns. For data placement policies, the information is given by users through feeding the data placement information to the metadata servers; users can implement typical data placement policies, or define their own policies. For data replication, the client managed data replication schemes can be implemented by spawning the same data I/O to multiple data servers at the clients; by enabling inter-server communication (which is not the default setup), the users can implement the metadata server managed or data server managed data replication schemes. For data caching, we have enabled it on the data server local file systems; however, we have not implemented this feature on the PFS server and PFS client components, which is considered as the future work.

3.2. Abstraction of PFS Scheduler

Among the many proposed centralized or decentralized scheduling strategies in distributed storage systems, there are a large variety of network fabrics and scheduler deployment schemes. For instance, in [19], the schedulers are deployed on the *Coordinators*, which reside between the system clients and the storage *Bricks* in a *FAB* system [27]. In [14], the scheduler is implemented on a centralized proxy, which receives all the system I/O and dispatches them to the disks in the storage systems. In [28], the scheduling policies are deployed on the network gateways which serve as the data center portals to the clients. And in [20], the scheduling policies are deployed on the per-server proxies, which intercept I/O and virtualize the data servers to the system clients.

In our simulator, the system network is simulated with high flexibility, which means the users are able to deploy their own network fabric with the basic or user-defined devices. The schedulers can also be created and positioned to any part of the network. For more advanced designs, inter-scheduler communications can also be enabled in multi-scheduler simulations. The scheduling algorithms are to be defined by the PFSsim users, and APIs are exposed to enable the schedulers to keep track of the system status.

4. Simulator Implementation

4.1. Overview of PFSsim

Based on the abstractions illustrated in section 3, we have developed the parallel file system simulator PFSsim, based on the discrete event simulation framework OMNeT++ and the disk system simulator DiskSim.

We will explain the simulated systems in PFSsim by introducing an example system, as shown in Figure 1. The simulated system contains 3 PFS data servers, 1 PFS metadata server, 2 schedulers and 3 clients. The entire system is deployed in a LAN with a switch. We have modularized the real-world entities in PFSsim. The basic modules include the data server, metadata server, scheduler, client and the switch/router. The network cable is simulated by the channel components. Each module may be composed by multiple components. For instance, the data server is composed by 2 or 3 components, namely the data server daemon, the local file system (optional) and the disk system. As also shown in the figure, within the entire system, only the disk system component is simulated by the DiskSim simulator; all other modules/components are simulated by OMNeT++.

Now we are going to describe the processing of a typical client data access request in the simulated system. The client data access requests are provided in the form of trace files. Each piece of trace includes access time, file ID, index, data size, read/write, etc. Upon reading one I/O request from the trace file, a REQUEST object is created at the client. The client then tries to obtain the placement information for the target data. It first checks the local cache; if not cached, the client will query the metadata server by sending and receiving the QUERY messages. After that, the

REQUEST object may split the target data range to smaller ranges according to the data distribution information and the data packet size limit. The JOB messages, which contain the access requests to each data server, are created at the client. Instead of being sent to the data servers, the JOBs are first sent to the schedulers. At the schedulers, the JOBs are reordered according to the scheduling algorithm, and eventually sent to the data servers. When the data server receives a JOB message, the data server daemon component may first conduct a locking operation on the requested data, and forward the request to the local file system. The local file system maps the data range and file ID to the physical block numbers. The local file system also conducts the data buffering/caching. If a page is not found in cache or dirty pages need to be written back, disk access requests will be issued. The disk access requests are sent to DiskSim through an inter-process communication channel over a network connection (currently, TCP). When the block access request is accomplished on DiskSim, the finish time is sent back to the local file system component.

When the requested data access is done, the local file system hands it over to the data server daemon. The data server daemon may release the lock on the data, and sends it back to the scheduler. The scheduler marks the finish of the JOB (queued JOBs may be dispatched), and forwards it back to the client. At the client, the timestamps in JOB are written to the output, and the corresponding data access is marked as done at the REQUEST object. The REQUEST is checked to see if more JOBs need to be issued. If all the data in the REQUEST are successfully accessed, the information in REQUEST is written to the output. Iteratively, the client starts to read the next trace.

4.2. Scheduler Implementation

The scheduler module is designed for easy and flexible implementation of scheduling algorithms; meanwhile, we also enable the inter-scheduler communication for users to implement collaborative scheduling schemes.

We provide a base class for the implementation of all scheduling algorithms. The algorithms can be realized by inheriting this class. The base class contains the following essential methods:

void jobArrival(JOB * job);
void jobFinish(JOB * job);
void getSchInfo(Message * msg);
void sendSchInfo(int ID, Message * msg);
bool dispatchJob(int ID, JOB * job);

The *JOB* objects are the JOBs referred in subsection 4.1. The *Message* objects are the packets defined by the users for exchanging the scheduling information between schedulers. *jobArrival* is called when a new JOB arrives at the scheduler. *jobFinish* is called when the scheduler receives a finished JOB. *getSchInfo* is called when the scheduler receives a scheduler-to-scheduler message. *sendSchInfo* is called when the scheduler wants to send message to other schedulers. And *dispatchJob* is called when the scheduler wants to dispatch a JOB to a data server.

The simulator users can overwrite these methods to specify the specific behaviors. Also, more methods and data structures can be implemented to realize the customized scheduling schemes. Note that although the inter-scheduler communication schemes are featured, we did not evaluate them in this paper.

4.3. Network Implementation

In PFSsim, the network connections between entities are simulated by the channel components in OMNeT++. By setting the bandwidth, delay and packet error rate of the channels, users are able to simulate the network links.

The JOB objects referred in subsection 4.1 are the network packets when they are transmitted in the simulated network. Users can define the maximum size of JOBs, thus, the JOBs can be larger than the typical network packets. To avoid complexity and promote simulation efficiency, detailed models of real-world network protocols are not enabled by default. The impact of these approximations is partially shown in the results from section 5.2, but in most cases, it is shown to be negligible. If higher accuracy is demanded in the network simulation, users are able to extend the system with the INET framework [22] which supports many basic models for wired/wireless network protocols. Lower simulation efficiency may be expected in this approach.

PFSsim also contains a router/switch component, which is responsible for forwarding the packets to the destinations. Users are able to configure the delay of these network devices. For simplicity, in the simulated system, the entities are named and resolved by the static user-defined IDs. Thus, packets are routed/ switched according to their destination IDs.

4.4. Local File System Implementation

In PFSsim, the local file system component in the data server module has two major features: data address mapping and data caching/buffering.

We avoided exploring much toward the address mapping technologies at the local file systems, for the reason that 1) generally, file system block allocation heavily depends on the context of storage usage (e.g., EXT4 [29]), which is very dynamic; 2) we consider random disk seeking time as a less important factor in typical parallel file system environments compared with the significant factors such as total disk access time and network delay. In PFSsim, we assign the disk space to different files in a sequential manner.

The memory is simulated for the local file system to conduct data caching/buffering for disk read/write operations. Users are able to define the memory size and page replacement policies. On a page fault or dirty page write back, the local file system needs to have disk data access transactions with the disk system simulator, which is DiskSim.

4.5. Synchronization between OMNeT++ and DiskSim

In PFSsim, there is one OMNeT++ process (for the major framework) and multiple DiskSim processes (one per disk system). Since DiskSim has the functionality of getting the time stamp for the next event, OMNeT++ can always proactively synchronize with every DiskSim instance at the provided time stamp.

Currently we have implemented TCP connections between the OMNeT++ simulator and the DiskSim instances. In this way, the simulator can be deployed in a cluster. Even though optimizations are done in improving the synchronization efficiency, we found the TCP connection cost is still the bottleneck of simulation speed. In the future work, we plan to introduce more efficient synchronization mechanisms, such as shared memory, but as a tradeoff, that approach does not support distributed simulations.

5. Validation and Evaluation

In this section, we are going to validate and evaluate the PFSsim simulation results against results measured in a real system. A PVFS2 system containing 4 data servers and 1 metadata server is used as the benchmark system. The benchmark system consists of a variable number of clients. On each data server node, we also deployed a proxy-based scheduler [20] that intercepts all the I/O traffic on the local machine. All the nodes are built on a set of Xen virtual machines hosted on a cluster of eight DELL PowerEdge 2970 servers. Each virtual machine is configured with 2.4GHz AMD CPU and 1 GB memory. All virtual machines run para-virtualized 2.6.18.8 kernel with Ubuntu 8.0.4. EXT3 is used as the local file system for PVFS2 data servers.

The simulated system is configured with the measurements from the benchmark system. The cache access speed is gauged by 400MB sequential read/write operations on the VM cache, and the disk access speed is gauged by 6.4GB sequential read/write operations on the VM disk. According to PVFS2, content locking and caching are disabled in the data server daemon component. We set 900MB memory capacity for the local file system component (other processes possess an average of 100MB memory) with Least Recent used (LRU) page replacement policy. We set up a maximum dirty data size of 400MB, which is 40% of the 1GB total memory size (complies with the default *dirty_ratio* value in Linux). The block/page size is 4KB. The average network bandwidth between each client and data/metadata server is measured to be 1Gbps, and the corresponding network latency is 0.2ms. In the real system, schedulers and data servers are deployed on the same VM in a one-to-one manner. To simulate this, we deploy one scheduler on each router-data server channel to intercept the packets and set the scheduler-data sever channel to be ideal.

We use the “Interleaved or Random” (IOR) trace generator [30] to generate the benchmark I/O to the real system. This benchmark allows the specification of I/O patterns. We have developed a stand-alone trace generator for PFSsim, and with it we generated the PFSsim trace files in the same pattern.

5.1. PFS Simulator Validation

To validate the simulator fidelity under various system workloads, we have performed five independent experiments with 4, 8, 16, 32 and 64 clients for both read and write. Every client generates sequential read/write I/O to 400 files, each containing 1MB of data. The data of each file is evenly distributed to four data servers with the stripping size of 256KB. In order to evaluate the performance of I/O buffering /caching, the file reading tests are done on the same files right after the corresponding file writing tests, which means the data to read may still be in the memory due to buffered written content.

Figure 2 and Figure 3 depict the average system throughput and latency for various client configurations. Over all we can see that the simulated throughput and latency matches the real system measurements well. In the following we are going to discuss the results in details.

For the reading tests, with the setup of client number 4 and 8, the system provides high throughput. This is because the data are still in the cache due to the pre-conducted writing test. Thus, the majority of the data I/O is memory I/O which is very fast. We also observe that the throughput with the setup of 8 clients is twice as the setup of 4, which is due to the parallelism on the PVFS2 servers. For the tests with client number 16, 32 and 64, the reading throughput decreases dramatically. The reason is the VMs

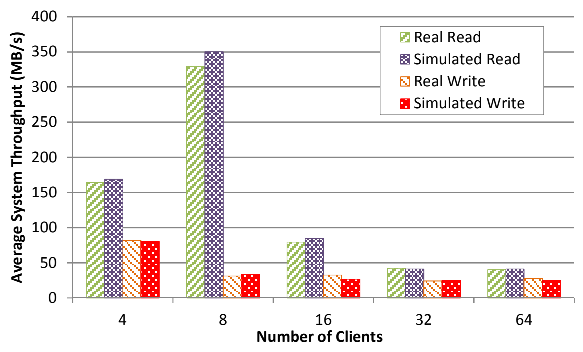


Figure 2. Average system throughput with different number of clients

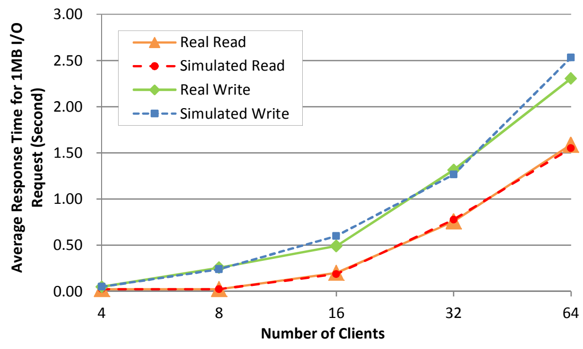


Figure 3. Average request response time with different number of clients

maintain limited amount of memory – the servers can never cache more than around 900MB of data. Thus, at least part of the data previously written to the servers is not in memory, and the read operations will incur the disk I/O rather than hit in the cache. One will notice that the throughput becomes stable when the client number is 32 and 64; the reason is almost all data are on the disk, so the saturated disk throughput becomes the system throughput.

For the writing tests, with the client number of 4, the written content can be mostly buffered in the memory, so it achieves higher throughput. But as the system client number increases, the throughput decreases because when the dirty data in memory exceeds 400MB, the system starts to write back the dirty pages, which incurs high penalty. Also, similar to the read tests, as the number of clients gets larger, the throughput gets more stable, because the saturated disk throughput determines the system throughput.

We can see from Figure 3, for 1MB I/O requests the simulated average response time matches the real system average response time well. The average response time grows non-linearly as the system workload increases. The reason is that the disk I/O delay is tens of times bigger than the memory access delay, and the queuing delay also prolongs the total delay when the data servers are under heavy load. From this set of tests, we can see that given the appropriate parameters, PFSsim is able to simulate a generic PFS system with an acceptable accuracy.

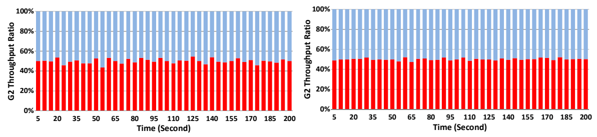
We also measured the simulation time in this series of experiments. We found that the major factor to the simulation time is the disk access time, because the TCP inter-process synchronization between OMNeT++ and DiskSim is very expensive. But even for the 64-client read test, it takes less than 3 minutes to finish on a PC (Intel Duo Core 1.66GHz CPU, 2GB Memory).

5.2. Scheduler Validation

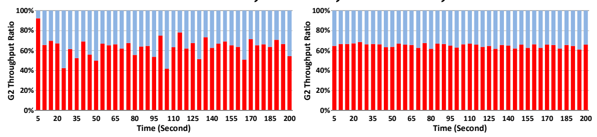
In this subsection, we are going to validate PFSsim in terms of the I/O scheduling algorithm simulation. We deploy 32 PVFS2 clients in the system. Each client issues sequential write I/O to 400 files, each containing 1MB of data. The data of each file is evenly distributed to four data servers with the stripping size of 256KB. For the purpose of algorithm testing, the clients are divided into two groups, G1 and G2, each with 16 clients. The Start-time Fair Queuing algorithm with depth D = 4 (SFQ(4)) [15] is deployed on each scheduler, which enforces the weight-based I/O proportional sharing.

We conducted three sets of tests, namely, set A, B and C. In SFQ(4), set A, B and C are configured with the weight ratios (G1:G2) of 1:1, 1:2 and 1:4, respectively. Every set is tested in both real system and the simulated system. We sampled the ratios of G2's throughput to the overall throughput during the first 200 seconds of runtime and calculated the averages and standard deviations for each set.

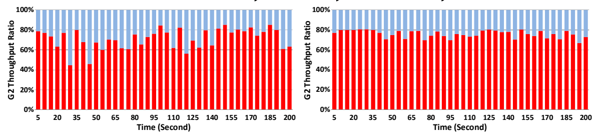
Figure 4 gives the pictures of G2's throughput ratios during the first 200 seconds of system runtime. The sample interval is 2.5 seconds. Calculated averages and standard deviations for each set are provided in the captions. First, we can see for the 3 sets, the averages of simulated throughput ratios are very similar (<5% error) to the averages from the real system. This means PFSsim is capable of simulating the major goal of SFQ(D) – I/O proportional sharing. Second, from both the real system and the simulation results, we see the standard deviations in the throughput ratios grow as the SFQ(D) algorithm applies a more imbalanced weight ratio. This is a special characteristic of the SFQ(D) algorithm. The increasing trend in the standard deviations from the simulation results provides helpful information to the algorithm designers. Third, we can also observe that the real system has a much higher standard deviation than the simulated system. This difference is due to the complexities existing in the real system, where many dynamic factors can contribute to the variations in the



Set A. 50.17%, 0.024, 50.06%, 0.011



Set B. 67.50%, 0.094, 65.26%, 0.019



Set C. 73.67%, 0.101, 76.63%, 0.038

Figure 4. The throughput ratios G2 takes in the first 200 seconds of runtime. In each row, the left chart is the real system result and the right chart is the simulation result. The caption format: set name, the average and standard deviation of G2's real system throughput ratios, the average and standard deviation of G2's simulated throughput ratios.

I/O throughput. Since PFSsim is using the abstracted models to simulate the real system performance, it is not be able to simulate all the details in the real systems. For example, TCP protocol is not simulated in PFSsim; also, the real-world network connections are not as stable as in PFSsim. However, as we are providing a prototype simulator, users are free to do more accurate simulations by extending it. It is also our future work to simulate the dynamics in the system with higher accuracy.

Overall, we can see PFSsim is able to simulate the performance trend of the SFQ(D) algorithm, while it also provides the system performance measurements with an acceptable accuracy. This information is very meaningful to an algorithm developer to evaluate the design before real implementations and evaluations.

6. Conclusion and Future Work

The design objective of PFSsim is to provide the users an easy-to-use, flexible, authentic and scalable PFS simulator for I/O scheduling algorithm designs. We provide a flexible scheduler module for scheduling algorithm deployment. The network topology, disk model, PFS specification and workload can be easily tuned. Since PFSsim has abstracted the major factors that contribute to the system I/O performance, we expect that given appropriate parameters, good simulation accuracy can be achieved. PFSsim is also highly extendable; users can extend any module of the simulator for higher accuracy or customized design. The validations on the PFS simulator and scheduler module show the system is capable of simulating the performance of a typical PFS system, given the PFS profiling parameters, scheduling algorithm and the workloads. For scalability, as far as we have tested, the system scales for simulations of up to 512 clients and 32 data servers. The simulator time efficiency is also shown to be acceptable.

In the future work, we are going to merge the IOR trace generator to PFSsim for generating synthetic traces. Meanwhile, more real benchmarks will be used for PFSsim evaluations. We will develop more accurate network models, which may characterize the statistical behavior of real TCP connections. Moreover, we are looking forward to simulate the disk systems with more abstractive models, which can substantiate DiskSim to promote the simulation efficiency. We are also investigating the possible ways to support simulations of very large scale systems. This may be achieved by allocating individual simulator modules into processes running on different nodes.

7. References

[1] E. Molina-Estolano, C. Maltzahn, J. Bent, and S.A. Brandt, "Building a parallel file system simulator", poster session presented in *SciDAC'09*, San Diego, CA, Jun. 2009.

[2] P. Carns, B. Settlemyer, and W. Ligon, "Using server-to-server communication in parallel file systems to simplify consistency and improve performance", in *Proc. the 2008 ACM/IEEE Conference on Super-computing (SC'08)*, Austin, TX, 2008, pp. 1–8.

[3] Sun Microsystems, Inc., "Lustre file system: high-performance storage architecture and scalable cluster file system", Sun Microsystems, Inc., Santa Clara, CA, white paper, 2008.

[4] P. Carns, W. Ligon, R. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters", in *Proc. the 4th annual Linux Showcase & Conference*, Atlanta, GA, 2000, pp. 317-327.

[5] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system", in *Proc. the 7th symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, 2006, pp. 307-320.

[6] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas activeScale storage cluster-delivering scalable high bandwidth storage", in *Proc. the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, Pittsburgh, PA, 2004, p. 53.

[7] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. "Understanding Lustre filesystem internals", Tech. Rep. ORNL/TM-2009/117, Oak Ridge National Lab., Oak Ridge, TN, 2009.

[8] R. Sandberg, “The Sun network filesystem: design, implementation, and experience”, Tech. Rep., Sun Microsystems, Mountain view, CA, 1987.

[9] M. Mesnier, G.R. Ganger, and E. Riedel, “Object-based storage”, *IEEE Communications Magazine*, IEEE Communications Society, Aug. 2003, pp. 84-90.

[10] Z. Dimitrijevic, and R. Rangaswami, “Quality of service support for real-time storage systems”, in *Proc. the International IPSI-2003 Conference*, Sveti Stefan, Montenegro, Oct. 2003, p. 20.

[11] R. Jain, K. Somalwar, J. Werth, and J.C. Browne, “Heuristics for scheduling I/O operations”, *IEEE Transactions on Parallel and Distributed Computing*, IEEE Press, Piscataway, NJ, Mar. 1997, pp. 310-320.

[12] F. Chen, and S. Majumdar, “Performance of parallel I/O scheduling strategies on a network of workstations”, in *Proceedings of the 8th International Conference on Parallel and Distributed Systems (ICPADS’01)*, KyongJu City, Korea, Jun. 2001, pp. 157-164.

[13] D. Durand, R. Jain, and D. Tseytlin, “Parallel I/O scheduling using randomized, distributed edge coloring algorithms”, *Journal of Parallel and Distributed Computing*, Academic Press, Inc, Orlando, FL, Jun. 2003, pp. 611-618.

[14] C.R. Lumb, A. Merchant, and G.A. Alvarez, “Façade: virtual storage devices with performance guarantees”, in *Proc. the 2nd USENIX Conference on File and Storage Technologies (FAST’03)*, San Francisco, CA, Mar. 2003, pp. 131-144.

[15] W. Jin, J.S. Chase, and J. Kaur. “Interposed proportional sharing for a storage service utility”, in *Proc. the joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS’04)*, New York, NY, Jun. 2004, pp. 37-48.

[16] 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*, IEEE Press, Piscataway, NJ, Oct. 1997, Vol. 5, pp. 690–704.

[17] J. Zhang, A. Sivasubramaniam, A. Riska, Q. Wang, and E. Riedel, “An interposed 2-level I/O scheduling framework for performance virtualization”, in *Proc. the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS’05)*, Banff, Alberta, Canada, Jun. 2005, pp. 406-407.

[18] A. Gulati, and P. Varman, “Lexicographic QoS scheduling for parallel I/O”, in *Proc. the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA’05)*, Las Vegas, NV, Jun. 2005, pp. 29-38.

[19] Y. Wang, and A. Merchant, “Proportional-share Scheduling for Distributed Storage Systems”, in *Proc. the 5th USENIX Conference on File and Storage Technologies (FAST’07)*, San Jose, CA, Feb. 2007, pp. 47–60.

[20] Y. Xu, L. Wang, D. Arteaga, M. Zhao, Y. Liu, and R. Figueiredo, “Virtualization-based Bandwidth Management for Parallel Storage Systems”. in *5th Petascale Data Storage Workshop (PDSW’10)*, New Orleans, LA, Nov. 2010, pp. 1-5.

[21] J.S. Bucy, J. Schindler, S.W. Schlosser, and G.R. Ganger, “The DiskSim simulation environment version 4.0 reference manual”, Tech. Rep. CMU-PDL-08-101, Carnegie Mellon University, Pittsburgh, PA, May. 2008.

[22] A. Varga, INET Framework for OMNeT++ 4.0, 2009. http://inet.omnetpp.org/.

[23] A. Varga, “The OMNeT++ discrete event simulation system”, in *Proc. the European Simulation Multiconference (ESM’01)*, Prague, Czech Republic, Jun. 2001.

[24] D. Roselli, J.R. Lorch, and T.E. Anderson, “A comparison of file system workloads”, in *Proc. the 2000 USENIX Annual Technical Conference*, San Diego, CA, Jun. 2000, pp. 41-54.

[25] S.A. Weil, K.T. Pollack, S.A. Brandt, and E.L. Miller, “Dynamic metadata managemnet for petabyte-scale file systems”, in *Proc. the 2004 ACM/IEEE Conference on Supercomuting (SC’04)*, Pittsburgh, PA, Nov. 2004.

[26] S.A. Weil, S.A. Brandt, E.L. Miller, and C. Maltzahn, “CRUSH: Controlled, scalable, decentralized placement of replicated data”, in *Proc. the 2006 ACM/IEEE Conference on Supercomputing (SC’06)*, Tampa, FL, Nov. 2006.

[27] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. “Fab: Building distributed enterprise disk arrays from commodity components”, in *Proc. the 11th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, Oct. 2004, pp. 48-58.

[28] D.D. Chambliss, G.A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T.P. Lee, “Performance virtualization for large-scale storage sytems”, in *Proc. 22nd International Symposium on Reliable Distributed Systems (SRDS’03)*, Florence, Italy, Oct. 2003, pp. 109-118.

[29] A. Mathur, M. Cao, and S. Bhattacharya. “The new EXT4 filesystem: current status and future plans”, in *Proc. the 2007 Ottawa Linux Symposium*, Ottawa, Canada, Jun. 2007, pp. 21–34.

[30] Interleaved or Random (IOR) Benchmark, https://asc.llnl.gov/sequoia/ benchmarks/IOR_summary_v1.0.pdf