Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–22

DOI: 10.1177/1094342016677084
journals.sagepub.com/home/hpc
SAGE

Rethinking key–value store for parallel I/O optimization

Anthony Kougkas[1], Hassan Eslami[2], Xian-He Sun[1], Rajeev Thakur[3] and William Gropp[2]

Abstract

Key–value stores are being widely used as the storage system for large-scale internet services and cloud storage systems. However, they are rarely used in HPC systems, where parallel file systems are the dominant storage solution. In this study, we examine the architecture differences and performance characteristics of parallel file systems and key–value stores. We propose using key–value stores to optimize overall Input/Output (I/O) performance, especially for workloads that parallel file systems cannot handle well, such as the cases with intense data synchronization or heavy metadata operations. We conducted experiments with several synthetic benchmarks, an I/O benchmark, and a real application. We modeled the performance of these two systems using collected data from our experiments, and we provide a predictive method to identify which system offers better I/O performance given a specific workload. The results show that we can optimize the I/O performance in HPC systems by utilizing key–value stores.

Keywords

Hyperdex, I/O performance, key–value store, parallel I/O optimization, performance evaluation, prediction model, OrangeFS

1 Introduction

File systems provide the interface between applications and the underlying storage space. A parallel file system (PFS) coordinates a large number of storage devices to serve applications' Input/Output (I/O) requests, leveraging the high degree of parallelism to offer the best I/O performance. For a given file, a PFS partitions the file into smaller fixed-size units of data, called stripes, and distributes them over multiple data nodes, according to predefined data layout policies. In the HPC community, PFSs seem to be the dominant storage solution and have served well for most of the requirement needs of an HPC system.

In the race for higher I/O bandwidth, PFSs have adopted a higher degree of parallelism. However, our previous work (Song et al., 2011b) has shown that a higher degree of parallelism might not always be better because data synchronization is more intense and could affect performance dramatically. A PFS's bandwidth can be largely affected by the file system's data layout policy (i.e. how data stripes are distributed physically) and the application's data access patterns (i.e. how an application reads or writes the data). The performance benefit achieved by parallel access brings an inevitable data synchronization issue, because different data nodes will perform differently and subrequests will finish at different speeds. The fast subrequests must wait for the slow ones, and the entire request finishes only after all subrequests are done. Additionally, most PFSs are designed to meet the POSIX standard, which requires a large number of metadata operations such as directory structure and file permissions. The latency caused by these metadata operations cannot be neglected. For a given application running over a PFS, its frequency and number of metadata operations can largely affect the overall I/O time. One example of a metadata-heavy workload is to read or write many small files (Carns et al., 2009). A PFS is often able to provide satisfactory bandwidth in terms of its design goal. However, because of factors such as data

[1]Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA
[2]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
[3]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

Corresponding author:
Anthony Kougkas, Department of Computer Science, Illinois Institute of Technology, Chicago, IL60616, USA.
Email: akougkas@hawk.iit.edu, sun@iit.edu

synchronization for unaligned requests and heavy-metadata operations, a PFS might demonstrate significant performance degradation with specific workloads. The experimental results presented in this paper have verified this phenomenon.

On the other hand, in large-scale cloud storage and web services, instead of file systems, key–value stores (KVStores) are being widely used. A KVStore provides an object based programming interface for manipulating the data. Each object is usually a key–value pair where the data are stored in the value and are represented with a unique ID, the key. The data read and write operations are presented as "get" and "put." Compared with PFS's fixed-size stripes, KVStore's object size is more flexible. Since an object is not further partitioned into smaller entities, there are no subrequests to synchronize between data nodes for an I/O operation to complete. Furthermore, hash tables often are used to manage the metadata (i.e. the mapping between the key and the physical location of the object). Different implementations of a KVStore keep various metadata information for each object. At the core of the key–value pair notion however, the number and frequency of the metadata operations are lightweight with low latency. Because KVStores are designed for high scalability, simplicity, and flexibility, they usually keep a simple flat namespace (not tree-structured). Each "put" or "get" operation usually comes with a fixed amount of metadata, including looking up the hash table or other structures being used and updating it when necessary, operations that are always faster compared with a PFS.

As we move toward the exascale era, when the data volume will explode, the shortcomings of the existing storage solutions in scientific computing will be even more challenging and efficient data handling will be critical. Based on this understanding, in this study we are not simply comparing the performance of these fundamentally different storage solutions by benchmarking them. Instead we aim to expose and identify the degrading factors in the performance of PFSs, as well as explore the potential use of KVStores that may not be sensitive to the same factors. We claim that for workloads that PFSs cannot handle well, a KVStore has a chance to achieve higher performance. We believe that it is time to loosen the reliance and dependence on the strongly restrictive POSIX standard and rethink the ways we handle data. Thus, we propose that KVStores be utilized to optimize the I/O performance for HPC systems. We encourage the reader to imagine a hybrid storage solution where PFSs and KVStores can complement each other and be used appropriately under scenarios that are better suited for them.

This paper presents our study of how we can optimize the I/O performance under an application's specific needs (i.e. workloads). We make the following contributions:

- We show that KVStore's performance stays stable with different data access patterns and various types of storage devices compared to PFS results (presented in Section 2). Our expectations for KVStore's potential for I/O optimization are verified by the results (presented in Section 3).
- We evaluate our proposal using synthetic benchmarks, a popular I/O benchmark, and a real application. The experimental results (presented in Section 4) show that KVStore, for certain workloads and system configurations, is able to provide higher I/O performance than a PFS does.
- We offer a performance prediction model (presented in Section 5) to choose the best-performing storage solution, a PFS or a KVStore, given the application's I/O characteristics and system configurations. We prove that our model is accurate in its predictions.

2 Motivation

PFSs have been around for a long time, and thus they have been optimized considerably. PFSs utilize massive parallelism to provide high-bandwidth data access to HPC applications. In some scenarios however, performance degradation can be experienced. This section presents two major performance degradation factors: (a) data synchronization, presented in Section 2.1, and (b) frequent metadata operation, presented in Section 2.2. In addition to these major factors, the type of storage device can also largely affect the I/O performance; we show comparison results in Section 2.3. All experiments in this section are performed on a Linux cluster with four nodes as storage servers. These nodes are equipped with both HDDs and SSDs and gigabit ethernet interconnection. The workload tested consists of only read operations with the data prepared beforehand (i.e. the necessary files copied in the PFS). We use OrangeFS (Carns et al., 2000; OrangeFS, 2014) as the PFS. We focus mostly on the raw I/O performance of the file system by measuring the time spent in I/O. More details on experiment methodology and the platforms used are presented in Section 4.

We present here some well understood issues of a PFS; our intention is to explain the motivation of our work to the reader and provide a clear demonstration of the problem we target to solve. We also provide quantifiable explanations of the performance degradation through a series of real system tests. Moreover, we showcase the scenarios in which there is a potential benefit to use a KVStore to optimize the I/O performance.

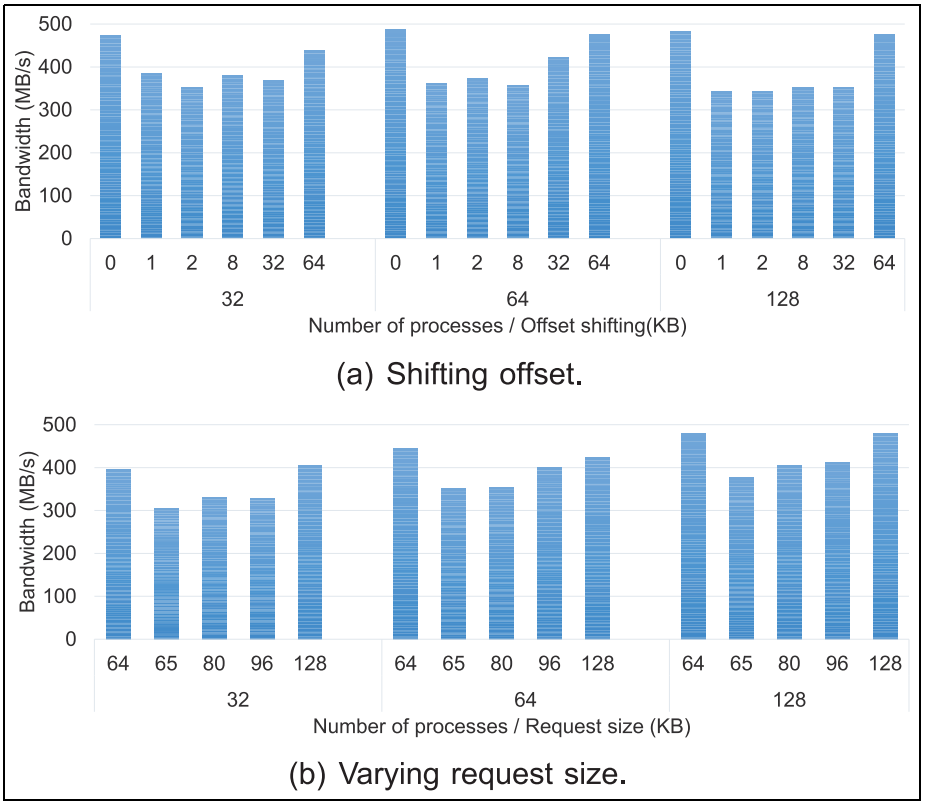

Figure 1. PFS with shifting offset / varying request size access patterns.

2.1 I/O performance degradation caused by data synchronization

To achieve parallel access, a PFS partitions the data into smaller fixed-size units of data called stripes and distributes them to multiple storage nodes. When an application requests some data, multiple storage nodes collaboratively service this request in parallel, hence the performance speedup. This collaboration introduces inevitable data synchronization and some issues that come with it. For example, assuming the stripe size is 64 KB, if a 64 KB request is aligned with the stripe boundary, then the storage node storing that stripe unit will serve the request by itself. If the request is not aligned with a stripe or is of size larger than 64 KB, then its data are distributed over more than one storage nodes. After the request is issued, all involved nodes work together to fulfill the request; each node delivers part of the demanded data via its corresponding subrequest. Because each storage node will act differently, the various subrequests will finish at different speeds. The fast subrequests must wait for the slow ones, and the entire request as a whole finishes only when all subrequests are done. This approach then, may cause severe overall performance degradation (Song et al., 2011b).

This negative effect on the I/O performance is caused mainly by the synchronization between storage nodes. We provide here some experimental results to demonstrate the degradation caused by data synchronization. In Figure 1(a), we repeatedly read a file by a 64 KB request size at various offsets. The PFS is configured with a 64 KB stripe size as its default value, and it is deployed on SSD devices. When the offset is 0 (or 64 KB in this case), each request is perfectly aligned with a data stripe; therefore, it involves only one storage node, and no data synchronization among subrequests is introduced. When the offset is 1 KB however, each request involves 2 storage nodes: 63 KB from one storage node and 1 KB from the next one; the same thing happens with 2 KB, 8 KB, and 32 KB offset shifting. We can clearly observe the performance degradation in Figure 1(a): 0 and 64 KB achieve the best performance, but the performance of all the other offsets is around 20% lower.

In Figure 1(b), we do not modify the offset, but we use different request sizes. We still use a 64 KB stripe size. When the request size is 64 KB, each request is perfectly aligned with a data stripe unit. With any request size that is larger than 64 KB however, each request inevitably involves more than one storage node. We can observe the obvious performance degradation in Figure 1(b): request sizes 64 KB and 128 KB yield the best performance, but all the other cases are 10 to 20% lower.

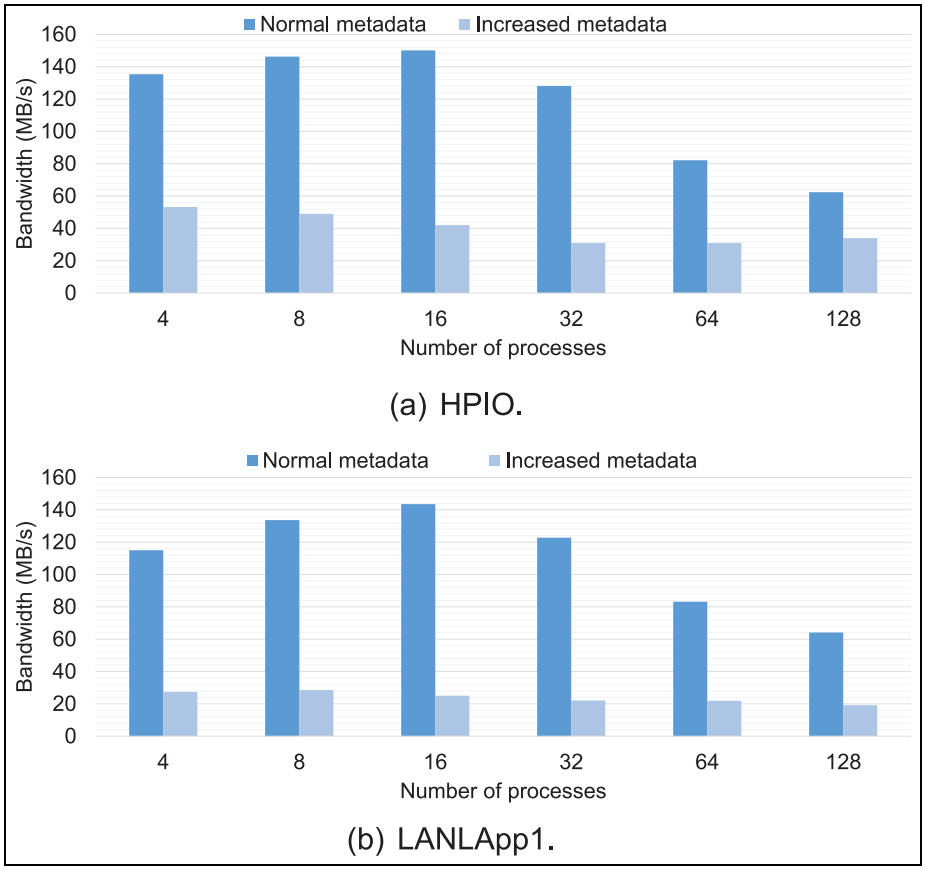

Figure 2. PFS performance without and with increased metadata operations.

2.2 Effect of metadata operations on i/o performance

Increased metadata operations can also significantly affect a PFS's overall I/O performance (Ali et al., 2008; Meshram et al., 2011; Ren et al., 2014). We provide here quantifiable proof of this effect. For our tests, the PFS's stripe size is configured at 64 KB and is deployed on HDDs. We generate the workload according to the I/O traces collected from two applications: HPIO (Ching, 2014) and LANLApp1 (Nuclear Engineering Division of Argonne National Laboratory, 2014). With the "normal metadata" test case, we open a large file, read the file according to the I/O patterns found in the trace file, and then close that file after performing all I/O operations. Each process opens and closes the file only once during this test. With the "increased metadata" test case, each I/O event in the trace file uses a separate file, a common access behavior where each process uses a different file (i.e. one file per process). In this case, each process performs one open and one close for each request. This test case stresses the PFS's metadata servers as expected.

From Figure 2, we can see that for both the HPIO and LANLApp1 workloads, the degradation on the overall I/O bandwidth, caused by increased metadata operations, is significant. Specifically, for 32 processes and normal metadata operation, OrangeFS achieves a bandwidth of 125 MB/s whereas with the increased metadata it drops to 32 MB/s, about 25% of the original bandwidth. This kind of degradation can be seen with various numbers of processes in both applications shown in Figure 2.

2.3 Comparison of PFS using HDD versus using SSD

SSDs generally offer better performance over HDDs. A PFS's performance can be seriously affected, though, especially in noncontiguous data accesses. Hardware resources must be utilized to the best of their capabilities and must avoid this degradation factor where possible. Figure 3 presents a comparison of the PFS's performance running the HPIO benchmark over different storage devices, HDDs and SSDs. At various numbers of processes the performance of HDDs is between 15% and 35% of what the SSDs achieve. Additionally, since metadata operations are latency sensitive, we can observe in Figure 3(b) that SSD devices are affected much less by the increased metadata operations.

The performance when using HDDs is, as expected, much lower than that when using SSDs, for several reasons. First, the physical raw performance of HDDs is

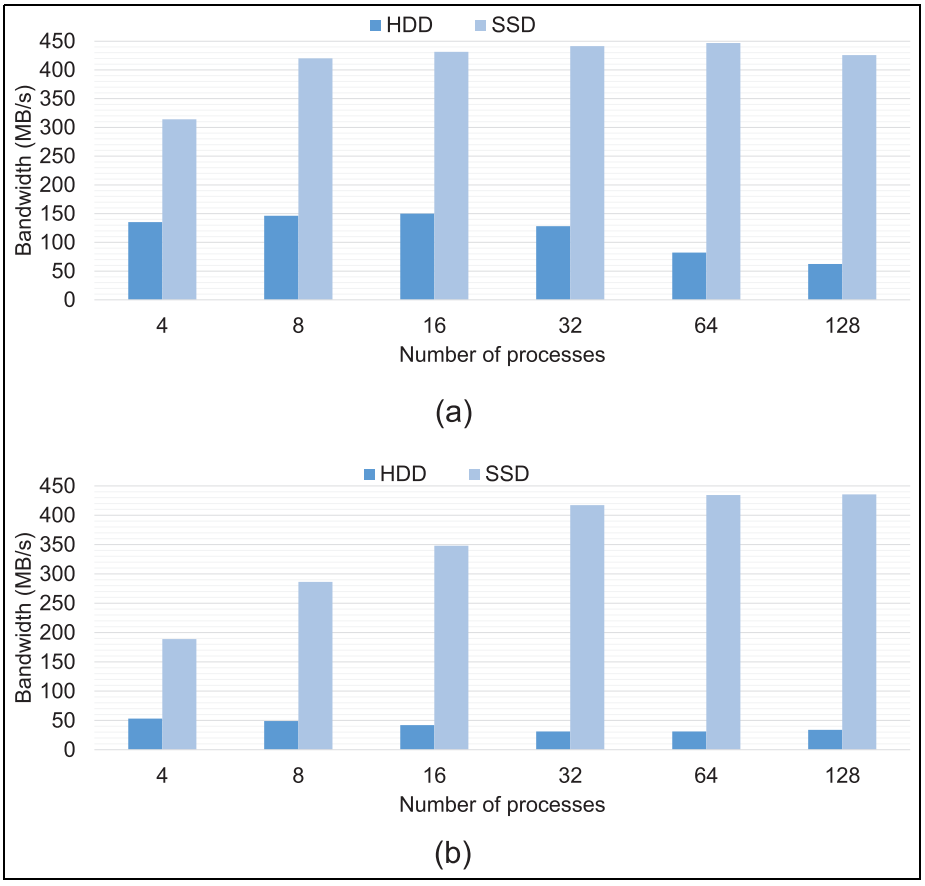

Figure 3. PFS with HPIO workloads on HDD and SSD: (a) with normal metadata operations; (b) with increased metadata operations.

lower. The rotational latency averages around 4 ms in modern HDDs, whereas SSDs roughly offer an access latency of around 0.1 ms (latencies measured by us on our testbed machine and may vary on different hardware). Second, data access patterns are important in the way the storage device is serving the requests. There are cases where each type of storage device performs well. For example, HDDs are good for streaming contiguous data accesses but perform badly if the data access is noncontiguous mainly because of the disk head seeking. On the other hand, SSDs are flash memory based and do not involve mechanical movement; thus they are much less sensitive to whether the data accesses are contiguous or not.

3 KVStore's potential in I/O optimization

PFSs provide satisfying data access bandwidth for HPC applications in most cases. However, a PFS's performance can largely degrade in certain scenarios, as shown in the previous section. We found that in these special cases, the performance of KVStores is much less affected. This motivates us to explore the opportunities of utilizing KVStores to optimize the I/O performance under those circumstances. This section presents the difference between PFSs and KVStores and also demonstrates KVStore's performance characteristics with experimental results.

We note the great diversity in the KVStore data management space. Implementations include distributed hash tables (i.e. DHT-based) such as Chord (Stoica et al., 2001), Dynamo (DeCandia et al., 2007), and ZHT (Li et al., 2013), or column-oriented implementations such as Cassandra (Lakshman and Malik, 2010) and MongoDB (MongoDB, n.d.), or even document-based implementations such as HBase (Vora, 2011). Most of them, however, in the core abstraction operate on the data in the key–value pair concept (i.e. object-based) and not in files. Many different features can be found in all of them, which are tailored for specific use cases. In this study, however, we focus only on the stripped-down version of how the raw data are being manipulated and whether that logic can be applied in a broader high-performance storage solution. The main question to be answered here is: Can we leverage the broader concept of a KVStore to provide better I/O performance to applications and alleviate the shortcomings of a PFS?

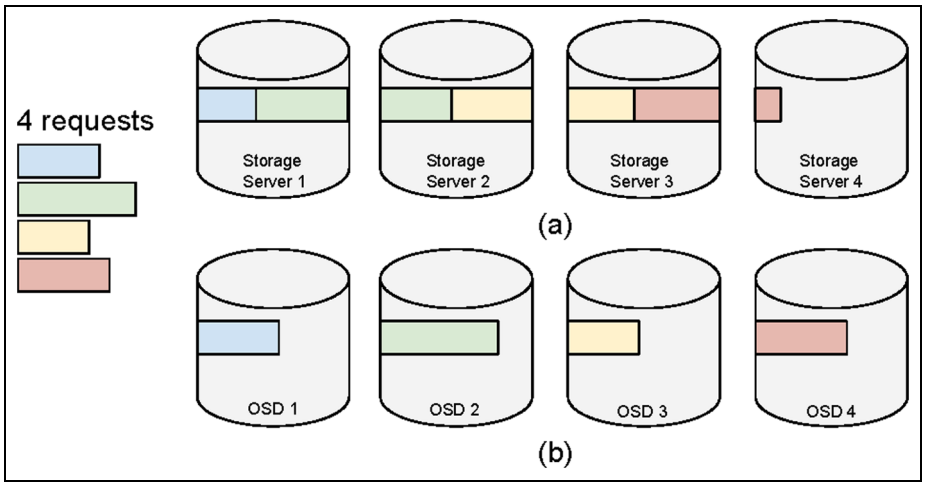

Figure 4. Stripe vs object on how the requests are physically stored and synchronized(boxes represent storage servers) (a) Data stored in a file in PFS (b) Data stored in objects in KVStore.

In all the experimental results presented in this section, we used our Linux cluster with four nodes running HyperDex (Escriva et al., 2012) as the KVStore servers; gigabit ethernet and SSDs were used; and the workloads were only read operations expressed as "get" operations. The data were placed as objects in the system prior to the execution of our microbenchmarks, which had as the main goal of capturing the raw I/O performance of HyperDex ignoring the extra features such as triple replication (disabled just to make sure) and searchable hyperspace for secondary attributes.

3.1 Architecture differences between PFS and KVStore

Both PFSs and KVStores are distributed storage systems and partition their data into small pieces that will be distributed over multiple nodes. However, the data partition and layout are different. Our goal is not to present all architectural differences here. Instead, we strive to demonstrate that for all the performance-degrading factors that affect a PFS, a KVStore would possibly perform better; and when examining the architecture benefits of each storage solution, we limit our discussion to investigating what makes this possible.

A PFS usually uses fixed-size stripes for a file, which are distributed in a fixed manner; the most widely used distribution policy is round robin. Figure 4 shows four requests and assumes they are contiguous data in a file. The PFS disregards the logical information of the requests. We can see in Figure 4(a) that the second, third, and fourth requests' data are each placed in two storage nodes. In Figure 4(b), however, KVStore treats each logical key–value pair as a single object and distributes all the objects to all available nodes; each object will not be further partitioned. The distribution is usually managed by an object-to-server mapping or hashing. Therefore, for each object only one server is involved, and internode data synchronization is avoided. In Figure 4(a), for each of the second, third, and fourth requests in the PFS, there is data synchronization among the two subrequests, whereas in Figure 4(b), for KVStore, there is none. The results in Section 3.2.1 verify that KVStore's performance is stable with different data access parameter "request size" and different data layout parameter "stripe size."

The difference in metadata management is that compared with a PFS, KVStore's metadata operation is more lightweight. The file layer metadata must include the directory tree, permissions for different users, and the data's physical location on disks. Various implementations of a PFS employ different techniques for handling the metadata workload, but typically the metadata operations can be a bottleneck. In contrast, KVStores were designed for scalability, simplicity, and reliability; and thus they usually maintain a flatter namespace with simpler structures that keep the mapping between keys and values. The experiments in Section 3.2.2 present KVStore's characteristics.

A PFS's performance can also be largely affected by the contiguity of the data access. The reason is that a file system usually takes advantage of the spatial data locality with data prefetching. This approach is especially important for HDDs. Thus regardless of the underlying device, the contiguous data access demonstrates high performance. On the other hand, with noncontiguous data access, the prefetching does not work well, and the disk head seeking in the case of a traditional spinning disk increases the data access latency. On the other hand, a KVStore manages a set of discrete data objects, and its performance usually does not benefit from any data locality. As a result, KVStore's performance does not vary largely with different access patterns or different storage devices. The results in Section 3.2.3 demonstrate this.

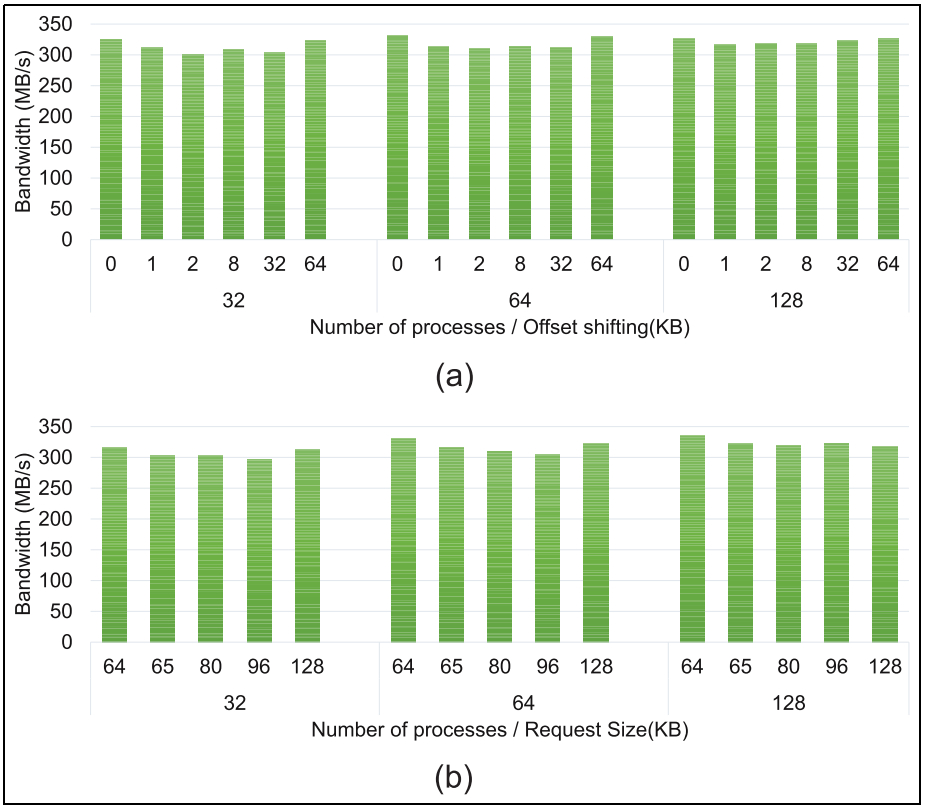

Figure 5. KVStore with different access patterns: (a) shifting offset; (b) varying request size.

3.2 KVStore's performance characteristics

This section illustrates how KVStore's performance varies with different workloads and different devices.

3.2.1 Stable performance with different access patterns. Figure 5 shows KVStore's performance with different workloads. The workloads are the same as that in Section 2.1. We vary the data offset, request size, and number of concurrent processes. This figure can be compared with Figure 1. We can see that compared with the PFS, KVStore's performance is stable no matter how these parameters change. However, the overall performance of KVStore is 300–350 MB/s, which is not as high as the PFS's nondegradation cases, 400–480 MB/s. The reason is that a PFS takes advantages of the data locality of contiguous access, while KVStore does not. Changing the representative storage solutions (e.g. another implementation of KVStore and of a PFS) would give different measurements. The important point here is not simply to compare the numbers but to note that the KVStore's I/O performance is not affected by the data access patterns and remains stable.

3.2.2 Stable performance with different metadata operation frequency. For a PFS, the frequency of metadata operations can vary largely, depending on how frequently the application creates directories, opens and closes files, and so on. But for KVStore, each put or get operation involves a fixed, smaller number of metadata operations, usually looking up or updating the structures that manage the mapping between keys and values. In both Figure 5 and Figure 6, the frequency of the metadata operation is one lookup per I/O request since it is a get operation. With a noncontiguous data access pattern and various request sizes in Figure 6 we observe a stable performance around 300 MB/s whereas the PFS in the similar microbenchmark achieved around 50 MB/s. In Section 4 we discuss the comparison between them in more detail.

3.2.3 Less degradation with slower disks. We wanted to investigate how KVStore utilizes the underlying storage devices. We ran HPIO and LanlApp1 workloads with various numbers of processes on HDDs and SSDs. In Figure 7, we can see that the performance with HDDs is 5–35% less than that with SSDs. This finding shows that object-based data access can make better use of the spinning disks than a PFS does. However, the overall SSDs bandwidth of around 300 MB/s is significantly less than that of the PFS bandwidth, around 450 MB/s. Under these specific workloads, a larger degree of parallelism achieved through the distribution of the stripes

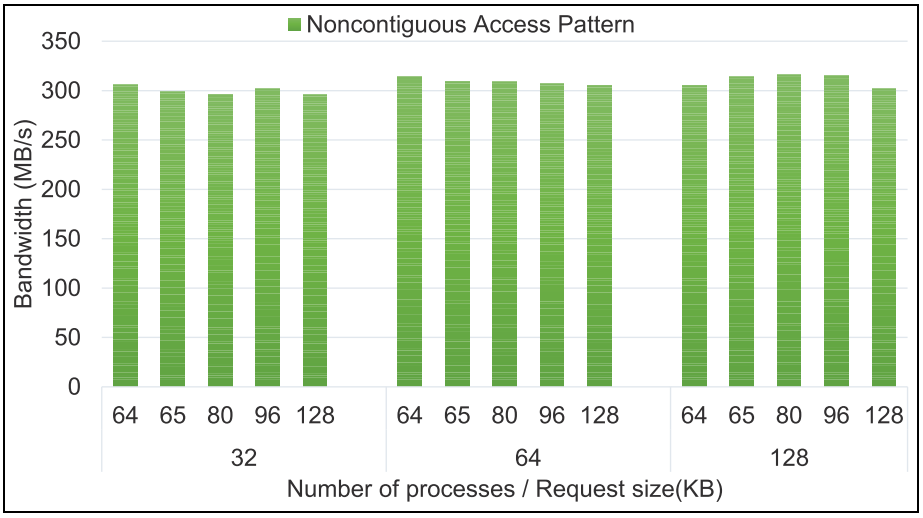

Figure 6. Metadata frequency stability.

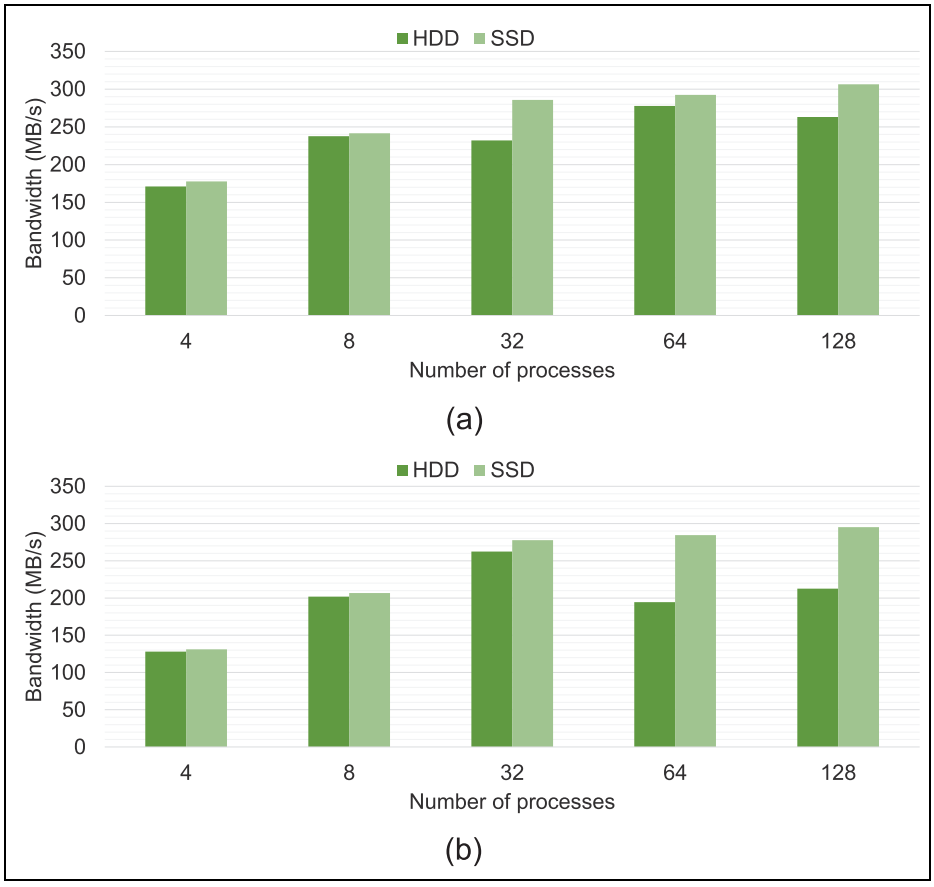

Figure 7. KVStore's performance with different storage devices: (a) HPIO; (b) LANLAPP1.

onto more than one SSD device leads to overall better performance. KVStore reads the data stored as a key–value pair from a single SSD device. Nonetheless, what we learn from this test is that the performance slowdown with HDDs is much less in the KVStore case.

In summary, comparing the results in Section 2 and Section 3, we make the following observations.

- With regular data access patterns with low data synchronization and relatively few metadata

operations, a PFS generates higher performance than does a KVStore.
- For workloads with irregular data accesses and heavy metadata operations, KVStore is expected to be a better choice than a PFS.
- KVStore performance demonstrates less variation with different storage devices compared with that of a PFS.

Based on these observations, we believe that it is valuable to utilize KVStore to optimize the I/O performance of some HPC applications, especially for workloads that do not favor a PFS.

4 Evaluation

To evaluate and further explore the potential use of KVStore systems in HPC environments, we conducted an extensive series of experiments.

Hardware specifications: Our testbed system is a 65-node SUN Fire Linux cluster. Each computing node has two AMD Opteron quad core processors, 8 GB memory, and a 250 GB HDD. All the nodes used are equipped with an additional PCI-E X4 100 GB SSD. All nodes are equipped with gigabit ethernet interconnection. The network topology of the cluster consists of three groups of nodes connected to network switches with the appropriate capacity (e.g. 22 nodes on a router of around 25 Gbits/sec) and a master node. A subgroup of nodes are connected through InfiniBand 4X network which we initially used to identify whether the network of the cluster would be a possible bottleneck. We found out that gigabit ethernet interconnection is sufficient to support our I/O benchmarks.

Software used: The operating system is Ubuntu 9.04, the PFS installed is OrangeFS v2.8.8, and the KVStore storage system is HyperDex v1.3. We compiled our code using gcc compiler version 4.8; the MPI implementation is MPICH 1.41b.

Experimental setup and configurations: For each set of tests, in OrangeFS we used four nodes both as storage servers and as metadata servers. For HyperDex, we also used four nodes as storage servers and a separate node as the coordinator (i.e. the node that controls all the metadata operations). The fact that HyperDex uses only one node as the metadata coordinator may seem like a lopsided situation; but KVStore involves fewer metadata operations and has a totally different architecture for metadata management. After benchmarking the metadata performance (which, due to lack of space, we do not present here), we have concluded that this setup is fair and would not be any kind of a bottleneck for any of those systems. We also made sure that client processes were located on different nodes from the servers (i.e. an entirely different network switch). This way, even though our testbed machine is a campus cluster, we managed to simulate a traditional HPC machine where computing nodes access the file system remotely.

Our choice of those specific storage systems as representatives from each category (e.g. file-based and KVStore-based) was made for several reasons. First, OrangeFS (formerly known as PVFS2) is a widely used PFS in the HPC community, and it is mature enough in terms of development and research to be the representative for the file-based storage system. HyperDex is a relatively new implementation by Cornell University. It is open source, and it has relatively easy to use APIs. It is well documented, and it has active support by its developers. According to Hyperdex, (n.d.) this KVStore implementation performs faster than other competitive KVStores. We acknowledge that different representatives of each storage solution may have differences in their implementations. For example, other PFSs, such as GPFS and Lustre, may have different implementation characteristics from OrangeFS; HyperDex may not behave exactly the same as other key value stores. However, the focus of this study is not benchmarking OrangeFS and HyperDex but the comparison of the two categories of storage solutions: data stripes-based PFSs and key–value-based object storage systems. We want to explore an instance of fixed versus variable storage segment sizes (PFS stripe vs KVStore object) and how existing assumptions might be revisited in the face of evolving use cases; OrangeFS and HyperDex were chosen since they implement the data stripes and key–value objects respectively. Although different representatives may lead to different measurements and numbers, we believe this issue does not hurt the conclusions and contributions of this study.

For the rest of this section, we first analyze our methodology and then present our experimental results. We do not present results in cases where a PFS performs well since our goal is to identify scenarios where a PFS performs badly and a KVStore can potentially offer higher bandwidth.

4.1 Methodology

Comparing the performance of OrangeFS and HyperDex under various scenarios is not an easy task since these two storage systems have different features and characteristics. Several factors can affect the performance of each system at any given time, such as data distribution schemes, data consistency, or fault tolerance guarantees. To achieve a fair comparison between them, we used the following method.

4.1.1 Tracing. IOSIG (Yin et al., 2012), an I/O pattern analysis tool developed at the I/O middleware level, is used to capture the runtime statistics of data accesses. Using this information, we were able to identify the

key characteristics of the I/O behavior of the application. The IOSIG trace includes information such as process ID, offset, request size, and begin and end time. We considered only the offset and the request size since these two values alone can determine the access pattern of the application.

4.1.2 Trace player/workload generator. Having the desired information extracted from the trace, we designed and implemented a straightforward workload generator. This workload generator takes an I/O trace as input and "replays" all the I/O operations onto the file system that is being tested. We developed three trace players; one for OrangeFS, one for HyperDex, and a third one for OrangeFS but modified to simulate extra metadata operations. The reason we designed this third trace player was to bring a balance between OrangeFS and HyperDex in terms of the amount of metadata produced by the systems. Traditionally, KVStore systems keep metadata for each object they store. On the other hand, OrangeFS operates on the same big file, which means that it opens the file once, does the I/O on this file, and then closes the file. The new trace player for OrangeFS, for each request to the file system for I/O, opens a small file, does the I/O, and then closes the file; but it does this for every request found in the trace. Thus, the amount of metadata produced by OrangeFS is similar to that from HyperDex. This new workload generator mostly simulates the behavior of OrangeFS when operating with many small files in applications such as graph applications. It is intended not to penalize the performance of OrangeFS but to emphasize some workloads that really hurt the performance and demonstrate the strength of the KVStore in similar cases. With these three workload generators, one can easily test the systems under various workloads; one simply feeds the I/O trace into the appropriate trace player and measures the time spent on I/O operations.

4.1.3 Performance measurement. To measure the performance of each storage system, we wrapped each I/O operation under a time barrier and calculated the total time spent in I/O. When doing so, special attention needs to be taken so that total time does not include other operations such as system startup or other preparations before the actual I/O operation. To focus on the file system performance, we removed the effects of memory cache and buffer. Before each test run, we cleared the operating system cache to ensure that all data was read from the storage devices. Prior to the first run, we also prepared the data for both systems. The experiments measure the read performance, and so the data was already in the underlying storage solution. For OrangeFS this was done with a simple copy, but for HyperDex we implemented a simple tool to copy the data into the storage system according to the trace of each application. Each request was eventually turned into an object with the offset as the key (can be up to 16 bytes) and the request size as the value (equal to request-size number of bytes). We ran each test 10 times and calculated the average time, leading the measurement closer to the actual time stripped from other factors that can degrade the system's performance, such as current system status, other running processes and overloaded network.

4.2 Results with synthetic benchmarks

We wanted to test specific access patterns that seem to affect the performance on a traditional PFS such as OrangeFS. In particular, we designed and implemented three simple synthetic benchmarks that can produce workloads with three distinct access patterns: offset shifting, varying request size, and noncontiguous access pattern.

4.2.1 Offset shifting. The first synthetic benchmark was designed to simulate an access pattern where the offset of the next request is shifted by some bytes (i.e. unaligned with the stripes of the PFS), thus forcing the system to coordinate each subrequest among multiple storage nodes. This specific access pattern clearly stresses OrangeFS; but it does not seem to be a problem for the KVStore system, where there is no need to synchronize the subrequests since each request is for a different object.

Figure 8 shows the comparison between OrangeFS and HyperDex for this offset shifting testing case. We captured the I/O trace of this synthetic trace and then gave it as input to the workload generator for each system. Additionally, we ran it with the modified OrangeFS benchmark with increased metadata operation. The results clearly illustrate that HyperDex can perform faster by an average of 244% and of 698% without and with the increased metadata operations respectively.

4.2.2 Varying request size. The second synthetic benchmark was designed to simulate a varying request size access pattern. The default value of the stripe size in OrangeFS is 64 KB. When a request is 64 KB or a multiple of that value, it is aligned with the stripes on each node. Generally, in PFSs, stripe sizes are fixed; and matching them with various request sizes is difficult. If a request is not aligned with the striping pattern, decomposition can make the first and last subrequests much smaller than the striping unit. This situation can lead to serious degradation, as we showed in Section 2. This benchmark stresses the storage system exactly according to that access pattern. We tried different cases where each process issues varied-sized requests

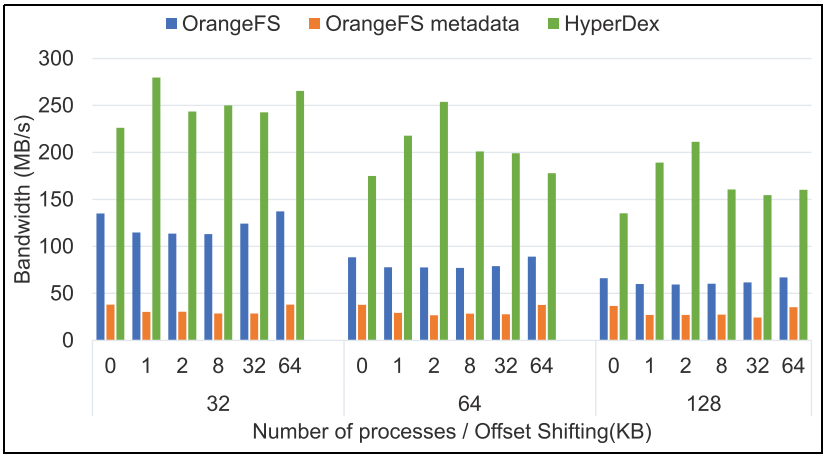

Figure 8. Shifting offset comparison.

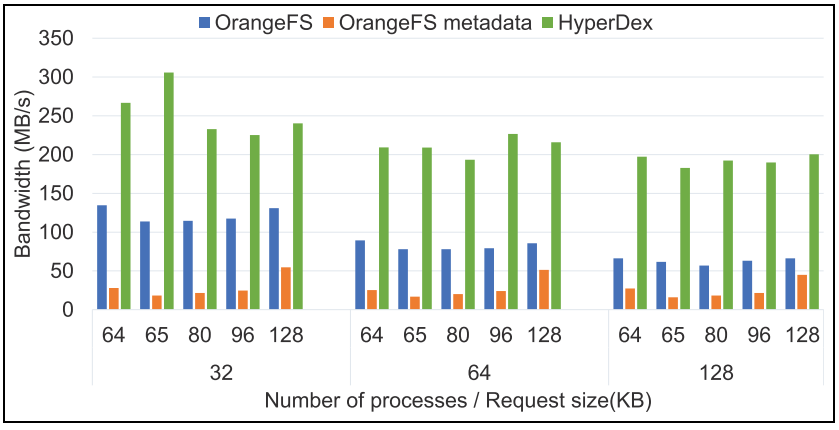

Figure 9. Varying request size comparison.

namely 64 KB, 65 KB, 80 KB, 96 KB, and 128 KB. The results can be seen in Figure 9. HyperDex clearly is not affected, and the performance is higher than OrangeFS by at least 183% and at most by 338%. We also gave the trace of this synthetic benchmark to the trace player with increased metadata operations, and the results are even more impressive. OrangeFS seems to suffer with this access pattern and also from the number of metadata operations. HyperDex achieves a higher bandwidth by an average of 914%. Specifically, this benchmark run on HDDs and HyperDex performs at around 200 MB/s, whereas OrangeFS with increased metadata operations is around 35 MB/s.

4.2.3 Noncontiguous access pattern. In this third synthetic benchmark, a noncontiguous access pattern is produced where a gap between each request is created, thus forcing the storage system to move across the file to do the requested I/O operation. Basically, each request is of various size and is served from various offsets inside the file.

Figure 10 illustrates the performance comparison between OrangeFS and HyperDex, first on HDD and then on SSD. We point out the difference that the type of storage device entails. In this case, when we load the trace in the workload generator and run it over HDD, the performance difference between these systems is large. Specifically, while HyperDex maintains a bandwidth of around 265 MB/s, OrangeFS is under 30 MB/s, and OrangeFS with increased metadata is even lower.

If we look at the SSD case, however, the picture is different. OrangeFS demonstrates good bandwidth and in some cases surpasses that of HyperDex. There is some obvious degradation for OrangeFS; with a 64 KB request size and 128 processes the bandwidth is close to 400 MB/s, whereas for 65 KB request size the bandwidth is 300 MB/s, a 24% degradation. Even with this degradation, however, OrangeFS manages to keep the performance high. Compared with HyperDex, which is very stable, OrangeFS has a 106% gain in average performance. OrangeFS with increased metadata operations demonstrates relatively low performance

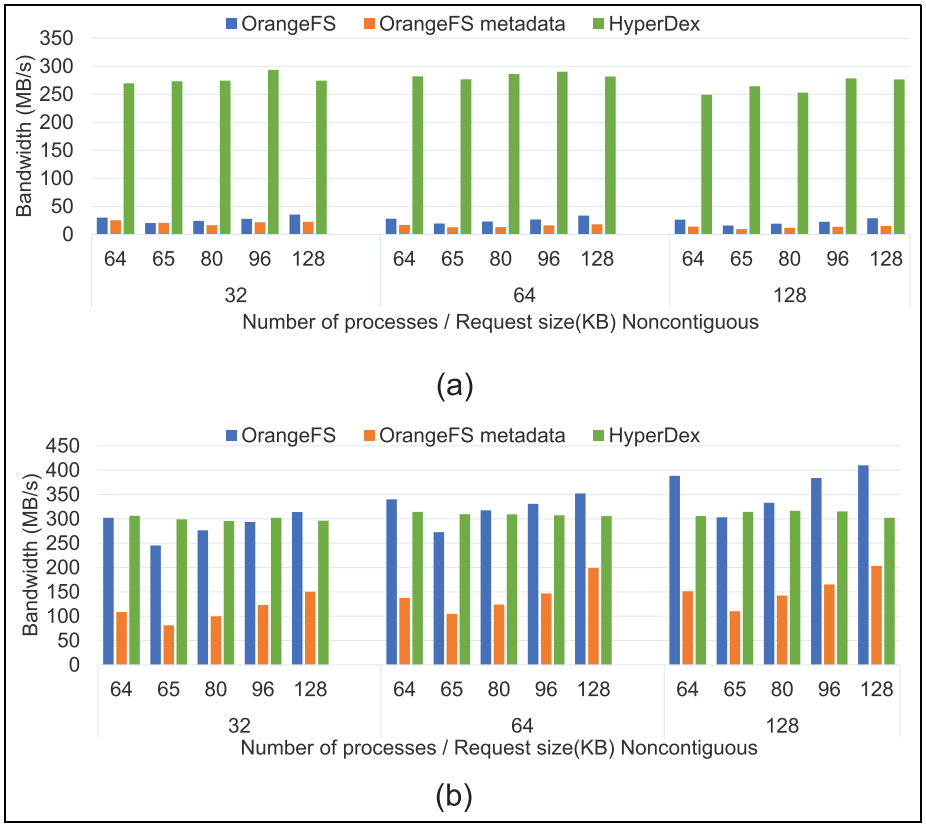

Figure 10. Noncontiguous access pattern comparison: (a) HDD; (b) SSD.

compared with that of the normal OrangeFS and HyperDex. In particular, HyperDex performs higher by an average of 238% compared with OrangeFS with increased metadata operations.

4.3 Results with HPIO benchmark

The HPIO (high-performance I/O) benchmark is a tool for evaluating and debugging noncontiguous I/O performance for MPI-IO. It allows the user to specify a variety of noncontiguous I/O access patterns and verify the output. It has been optimized for OrangeFS MPI-IO hints but can be augmented to use MPI-IO hints for other file systems. It is a widely used open source I/O benchmark designed and implemented by Northwestern University. We designed a noncontiguous access pattern with a fixed request size of 64 KB and measured the performance of these two storage systems.

Figure 11 demonstrates the comparison between OrangeFS and HyperDex. HyperDex's performance is consistently higher and scales well. It maintains an average bandwidth of 256 MB/s. OrangeFS seems to suffer from this particular access pattern; and as the number of processes increases, the performance decreases and reaches a low of 63 MB/s. HyperDex offers a better performance by an average of 241%. OrangeFS with increased metadata performs even worse, with a bandwidth of merely 50 MB/s.

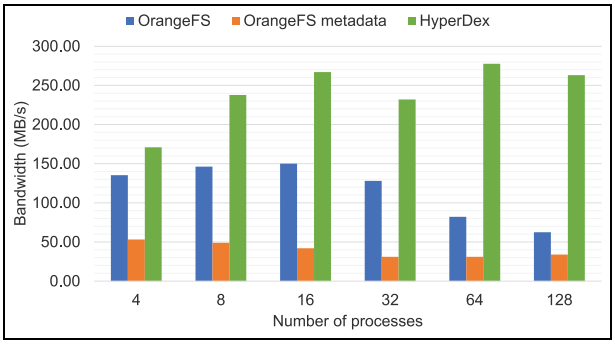

Figure 11. HPIO trace comparison.

4.4 Results with a real application

To test these numbers on real-world scientific applications, we took the I/O trace of LANL Anonymous App1 and fed it to the workload generator. This application, has three I/O requests in each loop, one small request with 16 bytes followed by two large requests with 131056 and 131072 bytes, respectively.

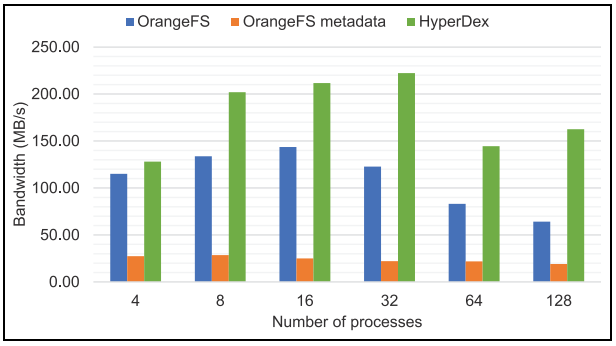

Figure 12. LANLApp1 trace comparison.

In Figure 12 we can observe that HyperDex hits a bandwidth of 220 MB/s with 32 processes, whereas OrangeFS is at only 120 MB/s. On average, HyperDex achieves a 179 MB/s bandwidth and OrangeFS 110 MB/s. When the increased metadata scenario was run, OrangeFS performs poorly, with a 25 MB/s average bandwidth resulting in an impressive 756% difference from HyperDex.

5 Performance toolkit

To further drive this evaluation comparison, we created a prediction tool to help users decide which of the two storage solutions would offer a higher I/O performance for their needs. In this section we present an overview of this tool, we detail the prediction model, and we provide some verification of the accuracy of the output of the model.

5.1 A performance prediction tool

To gather more information about each system's behavior, we decided to run more experiments with different configurations and the same trace files. Specifically we varied the number of available storage servers, the number of metadata servers, and the number of clients issuing the requests. We repeated all the test cases using the same traces, the ones from our microbenchmarks as well as HPIO and LANLApp1. The results were collected and created the dataset we used to build the performance prediction model.

The goal for this prediction tool is to be able to identify the best option in terms of higher I/O performance between the two storage solutions, PFS and KVStore. The inputs to this tool are the system configuration and the trace file collected by the IOSIG. Given an application and a system configuration, this tool captures the I/O characteristics, using the I/O profiling IOSIG, and determines the access patterns. Using this information, it runs the performance prediction model and returns the best-performing storage solution to use.

Table 1. Initial variables in the dataset.

Variable name	Variable description	Example entry
s	# of servers	4
m	# of metadata servers	4
c	# of clients	8
a	Access pattern (1-contiguous,2-noncontiguous)	1
g	Gap size between noncontiguous requests in bytes	0
r	Size in bytes of each request	262,144
p	# of processes	32
R	Total size in bytes for the input file	6,7108,864
d	Storage device type (1-HDD, 2-SSD)	2
A	Unaligned with the stripe boundaries requests (0-aligned, bytes-unaligned by these bytes)	0
M	Metadata operations (# of open/close/create operations)	768
f	Storage system (1-OrangeFS,2-HyperDex)	1
T	Time spent in I/O (us)	16,520,531

5.2 Model description

In this subsection, we describe the data and variables, present some preliminary analysis of the data, detail the multiple linear regression we used, and analyze the output.

5.2.1 Data and variables. Our dataset was derived from our extensive experiments and the file contained data from our performance measurements as well as the system configurations for each test case. Besides the time spent in I/O, we collected the access pattern information and included that in the dataset as well. The total number of entries in the dataset are 1329, representing all the conducted experiments. Each test case was repeated five times, and we kept only the average time for each one. We then categorized all the data into variables. Table 1 shows the initial variables in the dataset and an example entry.

The number of servers s refers to the number of available nodes to use as storage servers. The number of metadata servers m refers to the number of nodes acting as a metadata server or coordinator for HyperDex. The number of clients c is the number of clients issuing I/O requests to the storage solution. The access pattern a is a categorical variable with value 1 for a contiguous pattern and value 2 for a noncontiguous access pattern. For the noncontiguous access patterns we included the gap numeric variable g, which is the gap in bytes between each request. The reason we included this gap in between requests is that we wanted to examine whether the read-ahead (default 128 KB) of

Table 2. Means by filesystem.

Filesystem	Mean	Std deviation	Std error of mean
OrangeFS	92.762694	111.075107	4.2377797
HyperDex	212.552191	196.600468	7.7652516

Table 3. Compare means.

Filesystem	Access pattern		Storage device		# metadata servers		Same amount of metadata ops
#	1	2	1	2	1	4	Average
OrangeFS	95.068966	81.123016	52.272727	120.57198	29.611111	108.573333	44.814553
HyperDex	186.002727	182.056130	138.617318	160.088717	129.658691	210.626432	183.046248

the POSIX I/O can play a significant role in the performance. A smaller gap between requests might be alleviated by this read-ahead operation, but a larger request makes the disk head move more to serve the request. The variable s describes the request size in bytes. The number of processes p refers to the number of concurrent processes running in the application. R is the total size in bytes that the application has as an input file to issue the requests to. The different types of storage devices (i.e. disks), are described by the categorical variable d. Value 1 stands for the spinning hard drives (HDDs) and value 2 for the newer solid state drives (SSDs). To capture the requests unaligned with the stripe boundaries, we included the variable A that takes values 0 if the request is aligned with the stripes or the number of bytes that the offset of the request is shifted from the stripe boundaries. For instance, if the request started 2 KB from the stripe boundaries, the value for this variable would be 2048. The next variable is the number of metadata operations performed, namely, open, close, and create. The categorical variable filesystem F in the dataset describes the storage solution that this entry is coming from, with value 1 corresponding to OrangeFS and value 2 to HyperDex. T is the time spent in I/O as measured from our experiments

Initially, we ran some descriptive statistics to check whether our dataset was in good shape for the models. We tried some simple descriptive statistics, first comparing means between the two storage solutions in terms of the created variable bandwidth B. Table 2 shows the values. We expected that the mean of the two systems would be similar to these values since we observed from the evaluation that OrangeFS shows a high fluctuation between the offered bandwidth according to the test parameters and thus the lower mean value. OrangeFS had a 92 MB/s average bandwidth and HyperDex around 210 MB/s. Since both systems were tested in exactly the same test environment, we conclude that HyperDex seems to be more stable in its I/O performance.

During the evaluation we proved that for specific workloads a PFS is prone to extreme degradation in performance. We cite three key findings from the evaluation.

- With irregular data access patterns a PFS generates lower performance than a KVStore.
- For workloads with heavy metadata operations, a KVStore is expected to be a better choice than a PFS.
- KVStore performance has less variation with different storage devices compared with a PFS.

We ran the *compare means* statistical test to discover whether those observations are true. In Table 3 we can see the difference in the mean values for those variables. Access pattern, metadata operations, and type of storage device, in terms of the bandwidth change, are compared for both systems. The values shown are the average bandwidth offered from those systems in MB/s. The results clearly show that our conclusions hold.

Specifically, for the access pattern, OrangeFS shows almost a 20% degradation in performance when it serves noncontiguous accesses. On the other hand, HyperDex has better performance and is more stable; when going from contiguous to noncontiguous access patterns, the average bandwidth is kept around 180 MB/s. For the storage device type we can see that when OrangeFS operates with SSDs, the performance benefit is high; the system offers 230% higher bandwidth. This is expected; but it also shows that when OrangeFS is running over traditional HDDs and has workloads that are not favorable to its design, the performance can be low. On the other hand, HyperDex and the KVStore approach in general shows that the system is not that sensitive to the device type, mostly because of the way that it issues requests to the underlying disk and how it

Table 4. Correlation matrix.

Filesystem		# servers	# metadata servers	# clients	Access pattern	Gap in bytes	# processes	Storage device	Metadata operations
OrangeFS	Pearson	−.062	.313	.413	−.062	−.148	.055	.319	−.459
	P-value	.104	.000	.000	.104	.000	.150	.000	.000
HyperDex	Pearson	.382	.357	.263	−.013	−.008	.122	.158	.005
	P-value	.000	.000	.000	.752	.845	.002	.000	.898

handles the parallelism. KVStore does not need to stripe data among the available servers/disks; rather it distributes the objects among them. That approach makes the accesses to the disk more straightforward, and there is no need to synchronize any subrequests since there are none. HyperDex showed a performance benefit of 115% on the better hardware device, namely, the SSDs. Metadata operations can be described by the metadata servers variable and by the average bandwidth for the same number of metadata operations. We can see that going from 1 metadata server to 4 for OrangeFS means almost a 400% increase in the bandwidth, whereas for HyperDex we see a more moderate increase of 162%, which means that OrangeFS is more sensitive to the metadata handling than HyperDex (as described in Section 3). Furthermore, when we look at the average bandwidth for the same number of metadata operations (e.g. create/open/close operations), HyperDex achieves 4× higher bandwidth than does OrangeFS, which we saw in the evaluation section cannot handle a large number of these operations (i.e. increased metadata scenarios). Next, we checked the correlations between the variables and the independent variable bandwidth as well as some descriptive statistics. We demonstrate in Table 4 the correlation matrix by filesystem in terms of the variable bandwidth. We note that the *correlation matrix* shows the correlation between an independent variable with the dependent variable bandwidth.

For this test, the null hypothesis H_0 is that there is no significant correlation between the two variables involved. We can see from the table, however, a strong positive linear correlation does exist between the number of metadata servers, the number of clients, and the type of storage device variables and the dependent variable bandwidth for OrangeFS, where all p-values are less than 0.05. Thus we reject the null hypothesis. Similarly, there is a strong negative linear correlation between the metadata operations variable and the dependent variable bandwidth where again the p-value is less than 0.05. For OrangeFS, access pattern (the gap in bytes variable has Pearson correlation −.148 with p-value .000) and the metadata operations as well as the storage device type demonstrate a linear correlation with the I/O bandwidth, and those variables are expected to affect the performance. On the other hand, HyperDex shows almost the opposite. The access pattern and metadata operations have the lowest Pearson correllation with very high p-values. That means we accept the null hypothesis; there is no significant correlation between these variables and the dependent variable bandwidth. The HyperDex bandwidth seems to be affected by parameters that make more sense such as the number of servers, the number of metadata servers, the number of clients, and the type of storage device, where all those variables have a positive linear correlation and p-values less than 0.05. We reject the null hypothesis for those; there is a significant statistical linear correlation between the variables involved.

After this preliminary dataset analysis, we move to the multiple linear regression models for both systems. The dataset is representative of the performance of these systems, and the variables we choose can capture the factors that affect the I/O performance. The next subsection describes the prediction models and presents formulas to be used to predict the offered bandwidth according to the given system configuration and I/O workload characteristics.

5.2.2 Multiple linear regression. Multiple linear regression is an extension of simple linear regression. It is used when one wants to predict the values of a variable based on the value of two or more other variables. The variable we want to predict is called the dependent variable; in our case this is the new calculated variable bandwidth B. The variables we are using to predict the value of the dependent variable are called independent variables. In our case, the independent variables are all the variables from Table 1 besides filesystem and time which are used to calculate the new variable bandwidth. The independent variable bandwidth B is computed as $p * R/T$, and it refers to the aggregate bandwidth using all processes. Multiple linear regression also allows us to determine the overall fit of the model and the relative contribution of each of the independent variables to the dependent variable.

When analyzing data using multiple linear regression, part of the process involves checking to make sure that the data can actually be analyzed by this approach. During the preliminary analysis of our dataset we made sure that all assumptions that are required by the

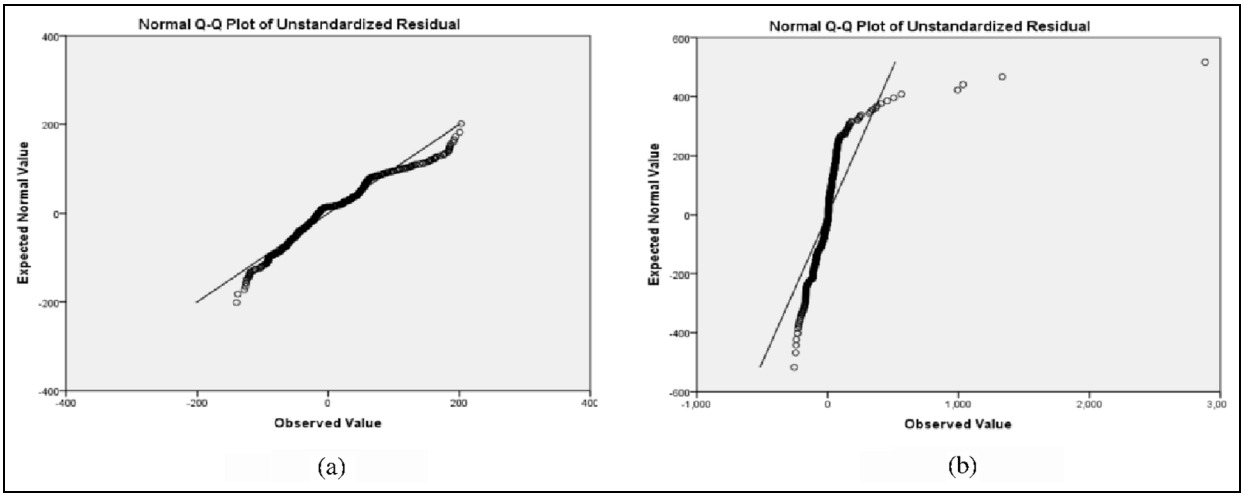

Figure 13. Q–Q plots of Residuals: (a) OrangeFS; (b) HyperDex.

multiple linear regression held. We do not list all eight assumptions here, but we will mention some. One assumption is that the dependent variable should be measured on a continuous scale, which holds for our bandwidth variable. Another assumption is that two or more independent variables are either continuous or categorical, which is also true in our case where we have 10 independent variables both numeric and categorical. Moreover, the residuals are approximately normally distributed for both models, as can be seen by the normal Q–Q plots of the residual values in Figure 13. After checking those assumptions we chose to run the multiple linear regression with the stepwise method on this initial model

$$B = b_0 + b_1 * s + b_2 * m + b_3 * c + b_4 * a + b_5 * g + b_6 * r + b_7 * p + b_8 * R + b_9 * d + b_{10} * A + b_{11} * M$$

where all b's are the coefficients of our independent variables and B is our dependent variable bandwidth (in MB/s).

We chose the stepwise method for our regression model because it evaluates the combination of the independent variables to best predict the dependent variable and thus gives us an advantage in finding out which independent variables each storage solution is more sensitive to. In stepwise regression, independent variables are entered into the regression equation one at a time based on statistical criteria. At each step in the analysis the predictor variable that contributes the most to the prediction equation in terms of increasing the multiple correlation, R, is entered first. When no additional predictor variables add anything statistically meaningful to the regression equation, the analysis stops. Thus, not all independent variables may enter the equation in stepwise regression. The next step is to run the multiple regression models with the stepwise method selected on the dataset with the selection variable filesystem: that is one regression model for OrangeFS and another one for HyperDex. The final models selected by the multiple linear regression for each storage system are

$$B_O = b_0 + b_1 * s + b_2 * m + b_3 * c + b_4 * a + b_5 * g + b_7 * p + b_8 * R + b_9 * d + b_{11} * M$$

for OrangeFS and

$$B_H = b_0 + b_1 * s + b_2 * m + b_3 * c + b_7 * p + b_9 * d$$

for HyperDex.

We note that the excluded independent variables for the OrangeFS model are the request size in bytes r and the unaligned bytes A. All other variables seem to be statistically significant and are kept in the final model. For HyperDex, on the other hand, the excluded independent variables are the access pattern a, the gap in bytes g, the request size in bytes r, the total size in bytes R, the unaligned bytes A, and the metadata operations M. Further explanations are provided in the next subsection. Table 5 summarizes the output of those models for each system.

5.2.3 Model analysis. We discuss a few important findings about the output of the multiple linear regression models we ran. First, both models show a good *model fit* with the adjusted R-squared value at .0633 for OrangeFS and .301 for HyperDex. Additionally, the p-values from the ANOVA table of both models are less than 0.05, which means that both models are statistically significant. Looking at the scatter plot and the histograms for the residuals in Figure 14 and given that the regression assumptions are satisfied, we conclude

Table 5. Regression models output.

		OrangeFS (model fit: Adjusted R-square=.633, ANOVA p-value=.000)						HyperDex (model fit: Adjusted R-square=.301, ANOVA p-value=.000)					
Variable Name	Coefficient	Beta	Std Error	t	p-value	95% lower	95% upper	Beta	Std Error	t	p-value	95% lower	95% upper
Constant	b_0	−177:033	27.210	−6:506	.000	−230:460	−.123:606	−282:618	30.890	−9:149	.000	−343:276	−221:959
s	b_1	−6:494	2.856	−2:274	.023	−12:101	−.887	61.754	6.566	9.405	.000	48.860	74.647
m	b_2	30.325	2.947	10.289	.000	24.538	36.112	−18:096	9.340	−1:937	.053	−36:438	.246
c	b_3	13.533	1.197	11.303	.000	11.182	15.884	12.283	3.057	4.018	.000	6.280	18.285
a	b_4	21.929	12.418	1.766	.058	−2:453	46.310	EXCLUDED					
g	b_5	−0:00023	0.00005	−4:685	.000	−0:0003	−.0:0001	EXCLUDED					
r	b_6	EXCLUDED						EXCLUDED					
p	b_7	.140	.057	2.454	.014	.028	.253	.592	.153	3.865	.000	.291	.892
R	b_8	6:702e−7	1:2e−7	5.543	.000	4:3e−6	9:0e−6	EXCLUDED					
d	b_9	69.034	5.044	13.686	.000	59.130	78.938	70.785	16.230	4.361	.000	38.914	102.657
A	b_{10}	EXCLUDED						EXCLUDED					
M	b_{11}	−.089	.010	−8:913	.000	−.109	−.069	EXCLUDED					

that the *goodness-of-fit* of those models is acceptable and we can trust that both models are adequate for predicting the dependent variable bandwidth.

Most important, we look at the excluded variables to explain the real behavior of the two storage systems. For OrangeFS, the final model kept all initial variables except two: the request size in bytes and the unaligned bytes. This shows that OrangeFS is more sensitive to the same parameters as we presented in Section 2. Specifically, OrangeFS I/O performance is severely affected by the access pattern, contiguous or noncontiguous. Metadata-heavy workloads also seem to affect the performance and the type of storage device is shown to fluctuate the bandwidth. This model tells us that to predict the bandwidth given the system configuration and the I/O trace, we need to take into account all variables kept in the model. Some of the variables, such as the number of the available storage servers and the number of processes, are expected to play a significant role in the final aggregate I/O performance. With this model, however, we are able to quantify how significant they are.

In contrast, the HyperDex model kept fewer variables, showing that the I/O performance of this system is less sensitive to parameters such as access pattern and heavy metadata operations. In this model, we need to use only four variables - the number of servers, the number of coordinators, the number of clients, and the type of the storage device - to predict the dependent variable bandwidth. Our observations from the evaluation for the potential use of KVStore for optimization of parallel I/O hold true here as well.

5.3 Verification of the model

To verify the accuracy of our models, we ran some new tests. We picked another scientific application from Los Alamos National Laboratory that offers the I/O trace (Nuclear Engineering Division of Argonne National Laboratory, 2014), anonymous application 2 (i.e. LANLApp2). Using the information from the trace file we extracted the access pattern, the gap between each request in bytes (since this application had a noncontiguous access pattern), the sizes of each request, the total size of the input file, and the number of metadata operations. We used in our testbed the same system configuration for both storage solutions. We used eight storage servers, four metadata servers, eight clients, 4–128 processes and repeated the experiments on both HDDs and SSDs.

Using all this information, we ran the two models and got a predicted bandwidth value for both systems. We then used the real machine and conducted the tests to measure the real I/O performance. The experimentation was conducted by using the same method as in the evaluation where we used our workload generators to

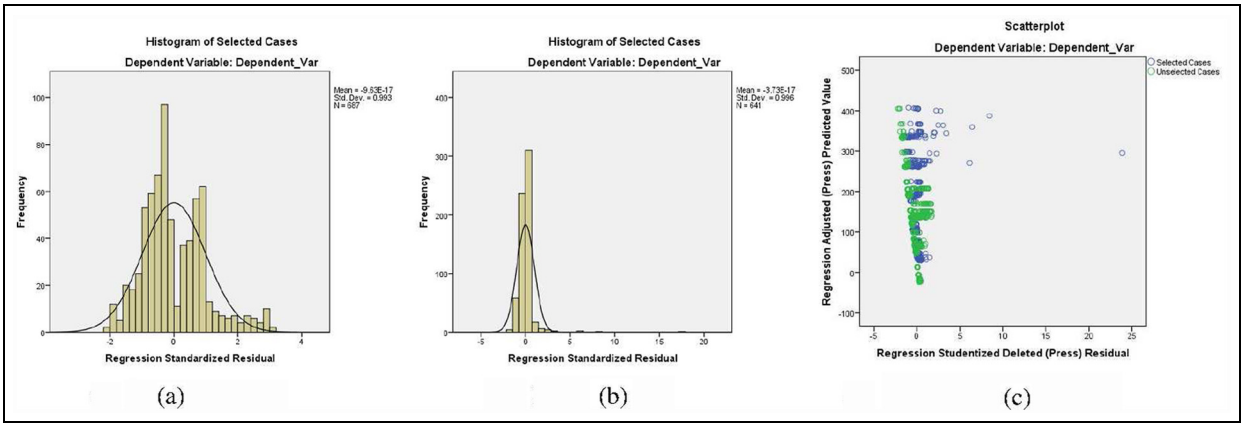

Figure 14. Standardized residuals for the regression models: (a) OrangeFS; (b) HyperDex; (c) Residuals scatterplot.

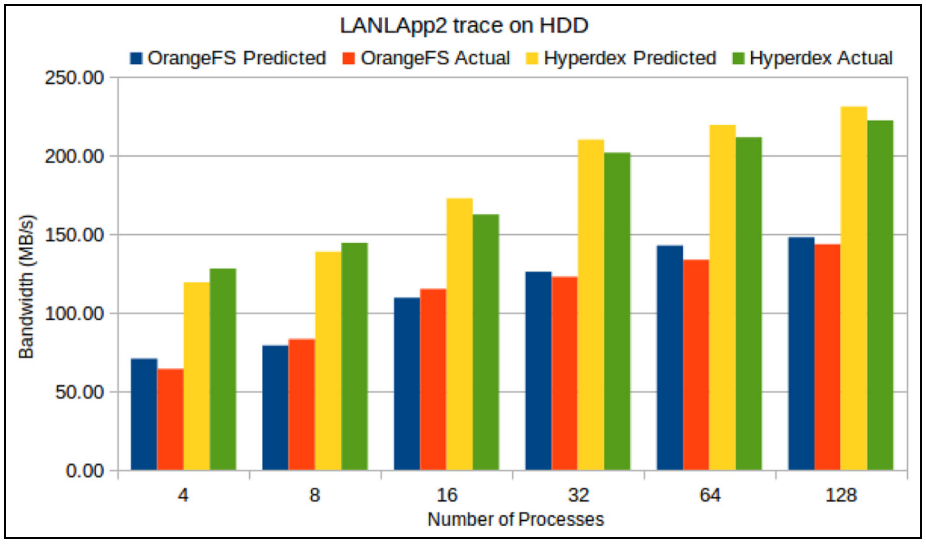

Figure 15. Predicted vs actual bandwidth for LANLApp2.

"replay" all I/O operations, and we measured the time spent in I/O. We then calculated the bandwidth using the total size in bytes and the measured time. In Figure 15 we can see that our predicted bandwidth value is close to the actual value from the real system. Our tool returns to the user the system with the higher expected bandwidth. As can be seen, HyperDex outperforms OrangeFS in this test. Thus, we can say that our models can successfully predict the aggregate bandwidth of the two storage solutions we are describing.

6 Related work

PFSs are the de facto method of data storage for HPC systems. They provide high data access performance and a consistent file based storage space. Popular PFSs include Lustre (Braam et al., 2014), OrangeFS, and GPFS (Schmuck and Haskin, 2002). Researchers found that the server-side data layout and client-side data access pattern can largely affect the overall I/O performance, because they affect the mapping between the logical data requests from the applications and the physical data layout on the server nodes (Song et al., 2011a). The parallel data access between the multiple client nodes and multiple server nodes inevitably brings data synchronization. Song et al. (2011b) designed an I/O coordination scheme to reduce the average completion time for concurrent applications. Zhang et al. (2013) noticed that the subrequest data synchronization caused performance degradation, and they designed a scheme called "iBridge," using SSDs to eliminate unaligned data access. The "offset shifting" and "varying request size" synthetic benchmarks are based on the benchmarks used in zhang2013ibridge's iBridge work.

Parallel and distributed file systems often decouple metadata management from I/O operations. However, metadata management services are often designed as centralized services for ease of implementation, which makes them hard to scale for billions of objects (Ghemawat et al., 2003; Shvachko, 2010; Shvachko et

al., 2010). This can be a bottleneck in metadata-heavy workloads because of namespace synchronization (Patil and Gibson, 2011). Modern file systems alleviate this problem by taking advantage of a distributed metadata service (Carns et al., 2000; Ren et al., 2014; Schmuck and Haskin, 2002; Wang et al., 2012; Welch et al., 2008; Weil et al., 2004, 2006; Zheng et al., 2014) by static or dynamic namespace partitioning among multiple servers. In particular IndexFS (Ren et al., 2014) uses a distributed KVStore to store metadata information.

Most PFSs support the POSIX standard. In many cases, however, POSIX is unnecessarily strict (Kimpe and Ross, 2014). It may hurt the system's scalability and the ability to control the small objects contained in a file independently (Goodell et al., 2012). Object-based systems and KVStores provide better flexibility with an object-based interface, instead of the file-based interface. KVStore is widely used in internet services and cloud storage services (Group, 2012), but it is rarely used for HPC systems. Many works have compared the advantages and disadvantages of file systems and object storage systems (Brim et al., 2013; Gibson et al., 1996; Group, 2012). Other works have tried to integrate PFSs and object storage systems (Devulapalli et al., 2007) or expose the underlying object data streams of a PFS (Goodell et al., 2012) to gain better I/O performance. This study explores whether HPC workloads can benefit from an object-based system such as KVStore.

Some work has been done on performance modeling and prediction for PFSs. Moody et al. (2010) use a probabilistic Markov model to predict the performance of a scalable checkpoint/restart. Sun et al. (2009) proposed a simple performance model to study integration of the parallel I/O middleware and PFSs. Using this model, the authors showed the effectiveness of data layout optimization in large-scale data storage. Nguyen and Apon (2012) used a colored Petri-net to measure and model the performance of PVFS. However, all these focus on modeling a single PFS, and they do not use their proposed model to compare different storage strategies.

7 Discussion

In this study, we use OrangeFS as the PFS and HyperDex as the KVStore. We know that the experimental results are dependent on the specific implementations of PFS or KVStore. So the actual performance of the tests presented in this study might change if they were run in a different platform or with different PFS and KVStore implementations. Still, we believe the comparison and the performance characteristics provided by the results are highly valuable.

8 Conclusion

PFSs are the dominant storage solution in HPC systems. Even though PFSs can offer a high-performance parallel data access, they have some deficiencies with specific I/O workloads such as being noncontiguous and unaligned with the stripe boundary access patterns. Heavy metadata operations are another difficult task for PFSs. On the other hand, KVStores are widely used by internet and cloud storage services, but are rarely used by HPC systems. Those storage solutions can offer an easy-to-use APIs for data manipulation and can offer competitive I/O performance. HPC systems can largely benefit from KVStores.

In this study, having compared the performance characteristics of PFSs and KVStores, we propose to utilize KVStores to optimize the performance for some specific I/O workloads, especially those that are difficult for a PFS to handle. We conducted extensive experiments the results of which prove the value of our proposal. We note that, we do not propose to entirely replace PFSs with KVStores. Rather we see them as complementary. A PFS's performance can be very high for its ideal workloads, but it can also be very low for some irregular workloads. With the same hardware, KVStore's performance is stable, somewhere between the PFS's high and low points. Therefore, it is valuable to optimize the performance with KVStore for the PFS's low-performance cases. We have proposed and implemented a performance prediction model to help users choose the best-performing system for their specific system configuration and application's I/O characteristics. The model is proven to be accurate, and using this toolkit can guide parallel I/O optimization.

In our future work, we want to extend the experimentation with better network interfaces such as InfiniBand and to improve our prediction model. We may evaluate some different representatives from PFSs and KVStores and examine their I/O performance behavior in order to generalize our proposal as much as possible. We feel encouraged to explore the general use of KVStores in HPC systems because we believe that applications currently running in these systems can benefit from this proposed data manipulation mechanism.

Acknowledgements

The authors acknowledge Harris Agyropoulos for his helpful insights in building our prediction model and Gail Pieper for the valuable text editing she provided.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the National Science Foundation (Grant numbers CNS-1162540, CNS-1162488, and CNS-1161507).

References

Ali N, Devulapalli A, Dalessandro D, et al. (2008) Revisiting the metadata architecture of parallel file systems. In: *Proceedings of the 2008 3rd petascale data storage workshop*, Austin, US, 15–21 November 2008, pp 1–9. Piscataway, NJ: IEEE.

Braam PJ, et al. (2014) The Lustre storage architecture. Available at: ftp://ftp.uni-duisburg.de/linux/filesys/Lustre/lustre.pdf. (accessed 13 February 2015).

Brim MJ, Dillow DA, Oral S, et al. (2013) Asynchronous object storage with QoS for scientific and commercial big data. In: *Proceedings of the 8th parallel data storage workshop*, Denver, US, 17–22 November 2013, pp. 7–13. New York, NY: ACM.

Carns P, Lang S, Ross R, et al. (2009) Small-file access in parallel file systems. In: *Proceedings of IEEE international symposium on parallel & distributed processing, IPDPS*. Rome, Italy, 25–29 May 2009, pp. 1–11. Piscataway, NJ: IEEE.

Carns PH, Ligon WB III, Ross RB, et al. (2000) PVFS: A parallel file system for Linux clusters. In: *Proceedings of the 4th annual Linux showcase and conference*. Atlanta, GA, USA, 10–14 October 2000, pp. 391–430. Berkeley: USENIX.

Ching A (2014) HPIO I/O benchmark. Available at: http://goo.gl/OEaaiB. (accessed 10 January 2015).

DeCandia G, Hastorun D, Jampani M, et al. (2007) Dynamo: Amazon's highly available key–value store. *ACM SIGOPS Operating Systems Review* 41(6): 205–220.

Devulapalli A, Dalessandro D, Wyckoff P, et al. (2007) Integrating parallel file systems with object-based storage devices. In: *Proceedings of the ACM/IEEE conference on supercomputing*, Reno, CA, USA, 10–16 November 2007, p. 27. New York, NY: ACM.

Escriva R, Wong B and Sirer EG (2012) HyperDex: A distributed, searchable key–value store. In: *ACM SIGCOMM computer communication review 42, no. 4 (2012)*, New York, NY, USA, pp. 25–36.

Ghemawat S, Gobioff H and Leung ST (2003) The google file system. In: *ACM SIGOPS Operating Systems Review*, volume 37, pp. 29–43. New York, NY: ACM.

Gibson GA, Vitter JS and Wilkes J (1996) Strategic directions in storage I/O issues in large-scale computing. *ACM Computing Surveys (CSUR)* 28(4): 779–793.

Goodell D, Kim SJ, Latham R, et al. (2012) An evolutionary path to object storage access. In: *High performance computing, networking, storage and analysis (SCC)*, Salt Lake City, UT, USA, 2012 SC Companion, pp. 36–41. Piscataway, NJ: IEEE.

Group TT (2012) EMC object-based storage for active archiving and application development. *Technical report, The TANEJA Group, Inc*. Available at: http://www.emc.com/collateral/analyst-reports/emc-atmosecosystem-taneja-group-tech-brief-final-ar.pdf.

Hyperdex (n.d.) Hyperdex performance benchmarks. Available at: http://hyperdex.org/performance/.

Kimpe D and Ross R (2014) Storage models: Past, present, and future. In: *High Performance Parallel I/O*, chapter 30. ANL, Lemont, IL, USA, pp. 335–345.

Lakshman A and Malik P (2010) Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44(2): 35–40.

Li T, Zhou X, Brandstatter K, et al. (2013) ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In: *Proceedings of the IEEE 27th international parallel and distributed processing symposium*, Boston, MA, USA, 20–24 May 2013, pp. 775–787. Piscataway, NJ: IEEE.

Meshram V, Besseron X, Ouyang X, et al. (2011) Can a decentralized metadata service layer benefit parallel filesystems? In: *Proceedings of the 2011 IEEE international conference on cluster computing*, Austin, TX, USA, 26–30 Septemeber 2011, pp. 484–493. Piscataway, NJ: IEEE.

MongoDB (n.d.) MongoDB. Available at: https://www.mongodb.com/white-papers. (accessed 2 April 2015).

Moody A, Bronevetsky G, Mohror K, et al. (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis*, New Orleans, LA, USA, 13–19 November 2010, pp. 1–11. Piscataway, NJ: IEEE.

Nguyen HQ and Apon A (2012) Parallel file system measurement and modeling using colored petri nets. In: *Proceedings of the 3rd ACM/SPEC international conference on performance engineering (ICPE '12)*, Boston, MA, USA, 22–25 April 2012, pp. 229–240. New York: ACM.

Nuclear Engineering Division of Argonne National Laboratory (2014) PLFS I/O traces. Available at: https://www.mcs.anl.gov/~thakur/pio-benchmarks.html (accessed 23 November 2016).

OrangeFS (2014) Orange file system. OrangeFS 2.8.8 Current version is 2.9.6 *Retrieved from:*http://www.orangefs.org.

Patil S and Gibson GA (2011) Scale and concurrency of giga + : File system directories with millions of files. In: *9th USENIX conference on file and storage technologies*, San Jose, CA, USA, 15–17 February 2011, Vol. 11, p. 13. Berkeley: USENIX.

Ren K, Zheng Q, Patil S, et al. (2014) Indexfs: Ccaling file system metadata performance with stateless caching and bulk insertion. In: *International conference on high performance computing, networking, storage and analysis*, New Orleans, LA, USA, 16–21 November 2014, pp. 237–248. Piscataway, NJ: IEEE.

Schmuck FB and Haskin RL (2002) GPFS: A shared-disk file system for large computing clusters. In: *Proceedings of USENIX conference on file and storage technologies*, Monterey, CA, USA, 28–30 January 2002. Berkeley: USENIX.

Shvachko K, Kuang H, Radia S, et al. (2010) The hadoop distributed file system. In: *Proceedings of the 2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp.1–10. IEEE. Incline Villiage, NV, USA, 3–7 May 2010, pp. 1–10. Piscataway, NJ: IEEE.

Shvachko KV (2010) Hdfs scalability: The limits to growth. *login* 35(2): 6–16.

Song H, Yin Y, Chen Y, et al. (2011a) A cost-intelligent application-specific data layout scheme for parallel file systems. In: *Proceedings of the ACM international symposium on high performance distributed computing*, Edinburgh, Scotland, 24–26 July 2010, pp. 37–48. New York/Piscataway: ACM/IEEE.

Song H, Yin Y, Sun XH, et al. (2011b) Server-side I/O coordination for parallel file systems. In: *Proceedings of the international conference for high performance computing, networks, storage and analysis (Supercomputing)*, Seattle, WA, USA, 12–18 November 2011, p. 17. New York/Piscataway: ACM/IEEE.

Stoica I, Morris R, Karger D, et al. (2001) Chord: A scalable peer-to-peer lookup service for internet applications. In: *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications* Review 31, no. 4 (2001), New York, NY, USA, pp. 149–160. New York: ACM.

Sun XH, Chen Y and Yin Y (2009) Data layout optimization for petascale file systems. In: *Proceedings of the 4th annual workshop on petascale data storage*, Oregon, Portland, USA, 14-2- November 2009, pp. 11–15. New York: ACM.

Vora MN (2011) Hadoop-hbase for large-scale data. In: *Proceedings of the 2011 international conference on computer science and network technology (ICCSNT)*, volume 1, Harbin, China, 24–26 December 2011, pp. 601–605. Piscataway, NJ: IEEE.

Wang Y, Zhou J, Ma C, et al. (2012) Clover: A distributed file system of expandable metadata service derived from hdfs. In: *Proceedings of the 2012 IEEE international conference on cluster computing* Beijing, China, 24–28 September 2012, pp. 126–134. Piscataway, NJ: IEEE.

Weil S, Brandt S, Miller E, et al. (2006) Ceph: A scalable, high-performance distributed file system. In: *Proceedings of 7th symposium on operating systems design and implementation*, Seattle, WA, USA, 6–8 November 2006, pp. 307–320. Berkeley: USENIX Association.

Weil SA, Pollack KT, Brandt SA, et al. (2004) Dynamic metadata management for petabyte-scale file systems. In: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Pittsburgh, PA, USA, 6–12 November 2004, p. 4. New York/Piscataway: ACM/IEEE.

Welch B, Unangst M, Abbasi Z, et al. (2008) Scalable performance of the panasas parallel file system. In: *6th USENIX conference on file and storage technologies*, San Jose, CA, USA, 26–29 February 2008, Vol. 8. pp.1–17. Berkeley: USENIX.

Yin Y, Byna S, Song H, et al. (2012) Boosting application-specific parallel I/O optimization using IOSIG. In: *Proceedings of IEEE/ACM international symposium on cluster, cloud, and grid computing*, Ottawa, Canada, 13–16 May 2012, pp. 196–203. Piscataway, NJ: IEEE.

Zhang X, Liu K, Davis K, et al. (2013) iBridge: Improving unaligned parallel file access with solid-state drives. In: *Proceedings of IEEE international parallel and distributed processing symposium*. Boston, MA, USA, 20–24 May 2013, pp. 381–392. Piscataway, NJ: IEEE.

Zheng Q, Ren K and Gibson G (2014) Batchfs: Scaling the file system control plane with client-funded metadata servers. In: *Proceedings of the 2014 9th parallel data storage workshop (PDSW)*, New Orleans, LA, USA, 16–21 November 2014, pp.1–6. Piscataway, NJ: IEEE.

Author Biographies

Anthony Kougkas is a 4th year PhD student at Illinois Institute of Technology in the Computer Science department, advised by Dr. Xian-He Sun. He holds a BSc in Military Science and a MSc in Computer Science, both received in Athens, Greece. His research is focused in Parallel and Distributed systems, Parallel I/O optimizations, HPC systems, and Key-Value Store solutions. He is a member of the Scalable Computing Software(SCS) lab at IIT and works closely with Argonne National Laboratory.

Hassan Eslami is currently a PhD candidate in the http://cs.illinois.edu/_blank Department of Computer Science at the University of Illinois at Urbana-Champaign, working with William D. Gropp. He received his bachelor's degree in computer engineering from Sharif University of Technology in 2011. His research focuses on Parallel and High Performance Computing. He is currently working on dynamic load balancing, and programming models for optimizing parallel I/O.

Xian-He Sun is the director of the SCS laboratory. He is a Distiniguished Professor of computer science and the past Chair (9/2009-8/2014) of the Department of Computer Science at the Illinois Institute of Technology, an IEEE fellow and a guest faculty in the Division of Mathematics and Computer Science at Argonne National Laboratory. His current research interests include parallel and distributed processing, memory and I/O systems, software system for Big Data applications, and performance evaluation and optimization.

Rajeev Thakur is a deputy director of the Mathematics and Computer Science Division at Argonne National Laboratory, where he is also a Senior Computer Scientist. He is also a senior fellow in the Computation Institute at the University of Chicago. He received a PhD in Computer Engineering from Syracuse University. His research interests are in the area of high-performance computing in general and particularly in parallel programming models, runtime systems, communication libraries, and scalable parallel I/O. He is a member of the Message-Passing Interface (MPI) Forum that defines the MPI standard for parallel programming. He is also a co-author of the MPICH implementation of MPI and the ROMIO implementation of MPI-IO, which have thousands of users all over the world and form the basis of commercial MPI implementations from IBM, Cray, Intel, Microsoft,

and other vendors. MPICH received an R&D 100 Award in 2005.

William Gropp received his BS in Mathematics from Case Western Reserve University in 1977, a MS in Physics from the University of Washington in 1978, and a PhD in Computer Science from Stanford in 1982. He held the positions of assistant (1982-1988) and associate (1988-1990) professor in the Computer Science Department at Yale University. In 1990, he joined the Numerical Analysis group at Argonne, where he was a Senior Computer Scientist in the Mathematics and Computer Science Division, a Senior Scientist in the Department of Computer Science at the University of Chicago, and a senior fellow in the Argonne-Chicago Computation Institute. From 2000 through 2006, he was also Deputy Director of the Mathematics and Computer Science Division at Argonne. In 2007, he joined the University of Illinois at Urbana-Champaign as the Paul and Cynthia Saylor professor in the Department of Computer Science. From 2008 to 2014 he was the deputy director for Research for the Institute of Advanced Computing Applications and Technologies at the University of Illinois. In 2011, he became the founding director of the Parallel Computing Institute. In 2013, he was named the Thomas M. Siebel chair in Computer Science. His research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. He has played a major role in the development of the MPI message-passing standard. He is co-author of the most widely used implementation of MPI, MPICH, and was involved in the MPI Forum as a chapter author for MPI-1, MPI-2, and MPI-3. He has written many books and papers on MPI including "Using MPI" and "Using MPI-2". He is also one of the designers of the PETSc parallel numerical library, and has developed efficient and scalable parallel algorithms for the solution of linear and nonlinear equations. With the other members of the PETSc core team, he was awarded the SIAM/ACM Prize in Computational Science and Engineering in 2015. Gropp is a Fellow of ACM, IEEE, and SIAM, and a member of the National Academy of Engineering. He received the Sidney Fernbach Award from the IEEE Computer Society in 2008, in 2010, and the SIAM-SC Career Award in 2014.