Rethinking Key-Value Store for Parallel I/O Optimization

Yanlong Yin*, Antonios Kougkas*, Kun Feng*, Hassan Eslami†, Yin Lu‡,
Xian-He Sun*, Rajeev Thakur§, and William Gropp†
*Department of Computer Science, Illinois Institute of Technology
{yyin2, akougkas, kfeng1}@hawk.iit.edu, sun@iit.edu
†Department of Computer Science, University of Illinois at Urbana-Champaign
{eslami2, wgropp}@illinois.edu
‡Department of Computer Science, Texas Tech University
yin.lu@ttu.edu
§Mathematics and Computer Science Division, Argonne National Laboratory
thakur@mcs.anl.gov

Abstract—**Key-Value Stores (KVStore) 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 (PFS) are the dominant storage systems. In this study, we carefully examine the architecture difference and performance characteristics of PFS and KVStore. We propose that it is valuable to utilize KVStore to optimize the overall I/O performance, especially for the workloads that PFS cannot handle well, such as the cases with hurtful data synchronization or heavy metadata operations. To verify this proposal, we conducted comprehensive experiments with several synthetic benchmarks, an I/O benchmark, and a real application. The results show that our proposal is promising.**

I. Introduction

Parallel file systems (PFS) are the dominant storage system used in HPC systems. A PFS usually manages a large number of storage nodes or devices and forms a global file system that is POSIX compatible. It aims to ease the I/O bottleneck of HPC applications by serving them with parallel data read/write operations over multiple nodes.

To achieve this parallel access, for a given file, a PFS partitions the file's data into data stripes using a fixed stripe size, and distributes those stripes over multiple data nodes, according to predefined data layout policies. A PFS's performance can be largely affected by file system's data layout policy (i.e., how data stripes are distributed physically) and application's data access patterns (i.e., how an application reads or writes the data). On top of that, the performance benefit achieved by parallel access, brings some inevitable data synchronization. Because different nodes may and will act differently, sub-requests may finish at different speed. The fast sub-requests must wait for the slow ones and the entire request finishes after all sub-requests are done.

Most PFSs are designed to meet the POSIX standard, which requires large amount of metadata such as directory structure, file permission, etc. The latency caused by metadata operation cannot be neglected. For a given application running over a PFS, its frequency and amount of metadata operations can largely affect the overall I/O time. One example of the metadata-heavy workload is to read or write many small files [1].

Most of the times, a PFS is able to provide satisfying bandwidth in terms of its design goal. But because of the above mentioned factors, such as data 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.

In large-scale cloud storage and Internet service systems, instead of file systems, Key-Value Stores (KVStore) are being widely used. KVStore provides an object based programming interface. Each object is usually a key-value pair. The data read and write operations are presented as "get" and "put." Hash tables are used to manage the metadata, i.e., the mapping between the key and the physical location of the object.

Compared with PFS's fixed-size stripes, KVStore's object size is more flexible. Since an object is not partitioned, there is no sub-requests for an I/O operation. KVStore is designed for high scalability and usually keeps a simple flat namespace (not tree-structured). The metadata operation of KVStore is light-weighted, with low latency. Also, the frequency of metadata operation does not vary, because each "put" or "get" always comes with a fixed amount of metadata operation, including looking up the hash table and updating it when necessary.

It can be seen that KVStore might not be sensitive to the performance factors that may severely degrade PFS's performance. Based on this understanding, we claim that, for workloads that PFS cannot handle well, KVStore may have a chance to grant a better performance. We need to rethink KVStore and utilize it to optimize the parallel I/O performance for HPC systems.

This paper presents our study on this proposal of using KVStore for parallel I/O optimization and makes the following contributions:

- We use experiments to demonstrate that PFS's performance can be largely degraded by unaligned data access, heavy metadata operation, and slower disks. The results (presented in Section II) are consistent with our expectations.

- We then conducted experiments to show that KVStore's performance stayed stable with different data access patterns and different types of storage devices. Again, the results (presented in Section III) verified our expectations on KVStore's potential on I/O optimization.
- We evaluated our proposal using synthetic benchmarks, a popular I/O benchmark, and a real application. The results (presented in Section IV) show that, KVStore is able to provide higher I/O performance than PFS does, for certain workloads.

Besides the above mentioned major sections, the other part of this paper is organized as follows: Section V presents the related work; Section VI discusses the uncertain issues in this study; and Section VII concludes the paper.

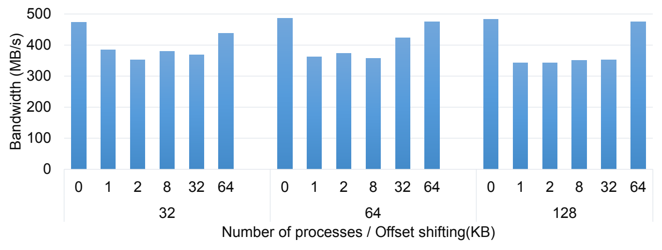
II. Motivation

Parallel file systems utilize massive parallelism to provide high data access bandwidth for HPC applications. However, in some scenarios, obvious performance degradation can be observed. This section presents two major causes of performance degradation: 1) data synchronization, presented in Section II-A, and 2) frequent metadata operation, presented in Section II-B. In addition to the above major factors, the type of storage device can also largely affect the I/O performance, we show some comparison on this in Section II-C.

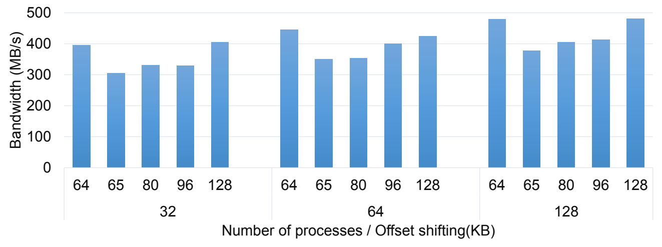
A. I/O Performance Degradation Caused by Data Synchronization

To achieve parallel access, a parallel file system partitions data into small-size data stripes and then distributes these stripes into multiple storage nodes. While an application requesting some data, multiple storage nodes collaboratively serve this request in parallel, and hence the performance speedup. This collaborative mechanism introduces the inevitable data synchronization. For example, assuming the stripe size is 64KB, if a 64KB request is aligned with a stripe, then the storage node storing that stripe will serve the request by itself. If this request is not aligned with a stripe, or its size is larger than 64KB, then its data are distributed over more than one storage nodes. After the request being issued, all involved nodes work together to fulfill the request; each node takes care of part of the demanded data of the corresponding sub-request. Because each node may act differently, the sub-requests may finish at varied speed. The fast sub-requests must wait for the slow ones and the entire request finishes when all sub-requests are done. This kind of waiting may cause severe overall performance degradation [2].

We demonstrate the degradation caused by data synchronization with some motivational experiments. All experiments are performed on SSDs with data read operations unless otherwise mentioned. We use OrangeFS [3] [4] as the PFS and HyperDex [5] as the KVStore. For more details on experiment methodology and platform, please check Section IV.



(a) Shifting offset



(b) Varying request size

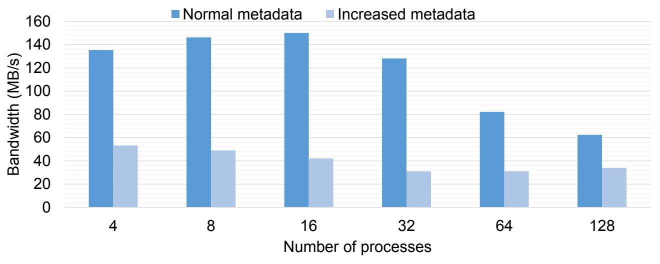
Fig. 1. PFS with Shifting offset / Varying request size access patterns.

In Figure 1(a), we repeatedly read a file with a 64KB request size but with various offset. The parallel file system is configured with a 64KB stripe size, while offset is 0 (or 64KB in this case), each request is perfectly aligned with a data stripe. Therefore, it only involves one storage node and no data synchronization among sub-requests is introduced. While the offset is 1KB, each request involves 2 storage nodes: 1KB from one storage node, 63KB from another; the same thing happens with 2KB, 8KB, and 32KB offset shifting. We can clearly observe the performance degradation from Figure 1(a): 0 and 64KB achieve the best performance but that of all the other offsets is around 20% lower.

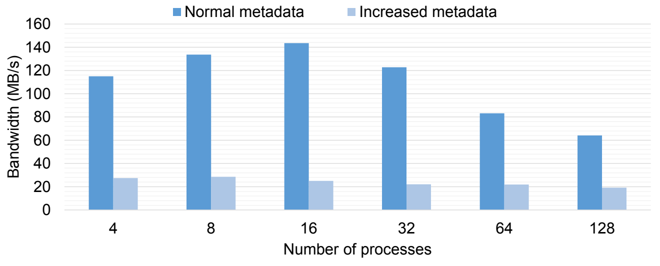
In Figure 1(b), we do not modify the offset but we use different request sizes. Still, we use 64KB stripe size. While the request size is 64KB, each request is perfectly aligned with a data stripe. With any request size that is larger than 64KB, each request inevitably involves more than one storage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

B. Metadata Operation Affects the I/O Performance

Increased metadata operations, can also significantly affect a parallel file system's overall I/O performance, as we can see in Figure 2. In this test, we generate the workload according to the I/O traces of applications, HPIO [6] and LANLApp1 [7]. With the "normal metadata" test, we open a large file and then read the file according to the trace files, and then close that file after performing all I/O. Each process opens and closes the file only once during each test. With the "increased metadata" test case, each I/O event in the trace file uses a separate file. In this case, each process performs one open and one close for each request. From Figure 2, we can see that, for both HPIO and LANLApp1 workloads, the degradation caused by

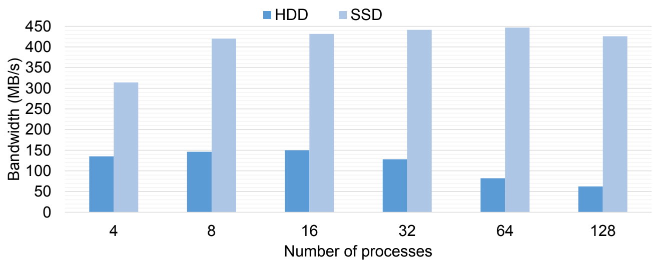


(a) HPIO

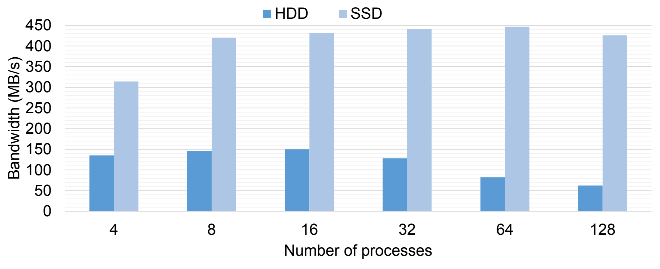


(b) LANLApp1

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



(a) With normal metadata operations



(b) With increased metadata operations

Fig. 3. PFS with HPIO workloads on HDD and SSD.

increased metadata operations is significant.

C. Comparison: PFS using HDD versus using SSD

It is widely known that SSDs offer better performance over HDDs. PFS's performance can be seriously affected from this fact especially in noncontiguous data accesses. It is important to utilize hardware resources to the best of their capabilities and avoid this degradation factor where is possible. Figure 3 presents the comparison of the PFS's performance running HPIO benchmark over different storage devices, HDDs and SSDs. The performance of using HDDs is much lower than that of using SSDs, for several reasons. First, the physical raw performance of HDDs is lower. Second, HPIO's data access is noncontiguous. Because of the disk head seeking, HDDs are naturally good for streaming/contiguous data accesses and performs badly for noncontiguous ones. SSDs are flash memory based and don't involve mechanical movement thus they are much less sensitive to whether the data accesses are contiguous or not.

In summary, this section shows some performance characteristics of PFS, including being sensitive to data access patterns, frequency and amount of metadata operations, and the type of storage devices. This section can serve as the motivation of this study, because Key-Value Store are much less sensitive to all these three factors (as presented in Section III). This gives us an opportunity to use Key-Value Store to optimize the performance for some specific workloads.

III. KVStore's Potential on I/O Optimization

Parallel file systems provide promising data access bandwidth for HPC applications most of the times. However, PFS's performance may degrade largely in some special cases, as shown in last section. We found that, in these special cases, the performance of Key-Value Stores is mush less affected.

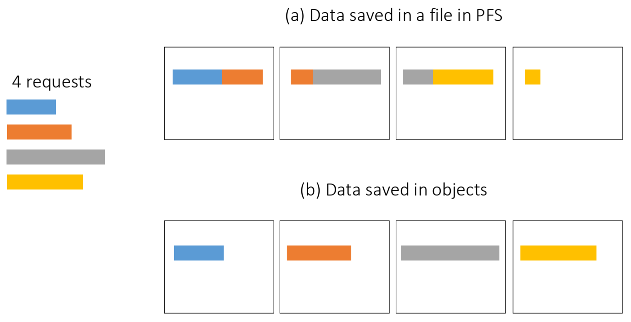


Fig. 4. Stripe VS Object on Data Synchronization

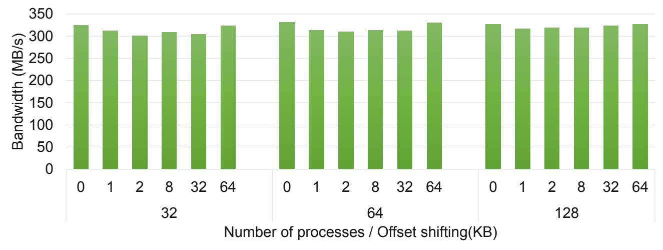
This motivates us to explore the opportunities of utilizing KVStore to optimize the I/O performance of these cases. This section presents the difference between PFS and KVStore and also demonstrates KVStore's performance characteristics with experimental results.

A. Architecture differences between PFS and KVStore

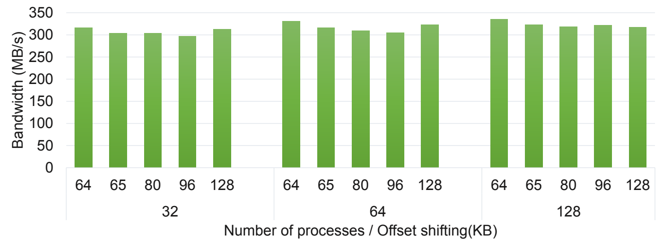
Both of PFS and KVStore are distributed storage systems and partition their data into small pieces that will be distributed over multiple nodes. However, the data partition and layout are different.

PFS usually uses fixed-size stripes for a file, and the stripes are distributed in a fixed manner. For example, the most widely used manner is the round robin. Figure 4 shows 4 requests and assumes they are contiguous data in a file. PFS uses fixed stripes for the file and disregards the logical information of the requests. We can see the second, third, and fourth requests' data are placed in two storage nodes.

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, as shown in part (b) of



(a) Shifting offset



(b) Varying request size

Fig. 5. KVStore with different access patterns.

Figure 4. The distribution is usually managed by a distributed hash table.

Different data layouts result in different data synchronization. In Figure 4, for each of the second, third, and fourth requests in PFS, there is data synchronization among the two sub-requests. For KVStore, there is none. The results in Section III-B1 verified that KVStore's performance is stable with different data access parameter "request size" and different data layout parameter "stripe size."

The difference on metadata management is that comparing with PFS, KVStore's metadata operation is more light-weighted. The file layer metadata must include the directory tree, permissions for different users, and data's physical location on disks. A KVStore usually maintains a flat namespace with a hash table that keeps the mapping between keys and values. The experiments in Section III-B2 present KVStore's characteristics in this aspect.

PFS's performance can also be largely affected by the contiguousness of the data access. This is because a file system usually take advantages of the spatial data locality with data prefetching. This is especially important for HDD. So, even with spinning disks, the contiguous data access demonstrates high performance. With noncontiguous data access, the prefetching does not work well and the disk head seeking will increase the data access latency. 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 III-B3 demonstrated this.

B. KVStore's Performance Characteristics

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

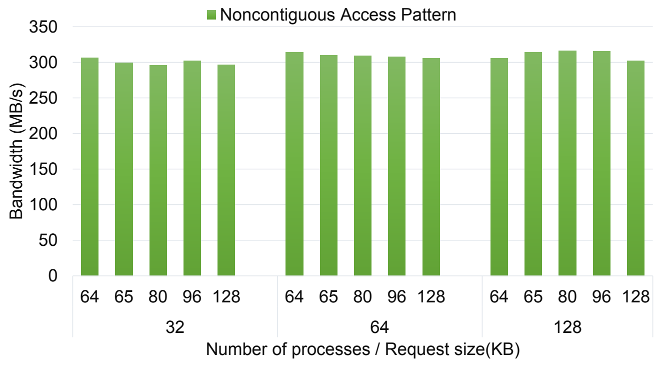


Fig. 6. Metadata Frequency Stableness

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 II-A. We vary the data offset, request size, and the number of concurrent processes. This figure can be compared with Figure 1. We can see that, comparing with PFS, KVStore's performance is very stable no matter how these parameters change. However, the overall performance of KVStore is 300-350 MB/s, which is not as high as PFS's nondegradation cases, 400-480 MB/s. This is because of that PFS takes advantages of the data locality of contiguous access, while KVStore does not.

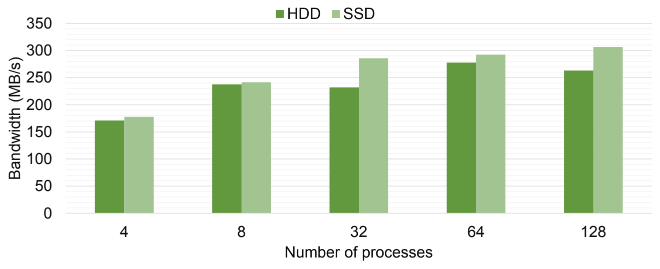
2) Stable performance with different metadata operation frequency: For PFS, the frequency of metadata operation can vary largely, depending on how frequently the application creates directories, opens files, etc. For, KVStore, each put or get operation has to involve one or more times of metadata operation, updating or looking up the hash table that manages the mapping between keys and values. In both Figure 5 and Figure 6, the frequency of metadata operation is one per I/O operation. We can see that, with the same workload, KVStore's performance (300-350 MB/s) is better than that of PFS with increased metadata operations (20-60 MB/s).

3) Less degradation with slower disks: It can be observed that different storage devices can result in different performance. With KVStore, the performance with HDDs is 5%-35% less than that with SSDs, as shown in Figure 7. The performance difference for PFS is 60%-85%.

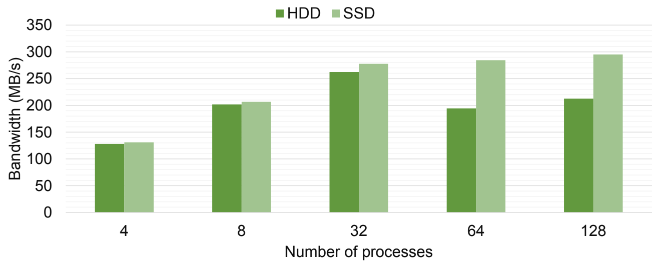
In summary, we have the following observations by comparing the results in Section II and Section III.

- With regular data access patterns with low data synchronization and light metadata operation, PFS generates high performance that KVStore cannot compete.
- For workloads with heavy metadata operations, KVStore is expected to be a better choice than PFS.
- KVStore performance has less variation with different storage devices comparing with PFS.

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



(a) HPIO



(b) LANLAPP1

Fig. 7. KVStore's performance with different storage devices.

IV. Evaluation

To evaluate and further explore the potential use of KVStore systems in HPC environments, we conducted an extensive series of experiments. Our testbed system is a 65-node SUN Fire Linux cluster. Each computing node has two AMD Opteron(tm) processors, 8GB memory and a 250GB HDD and all the nodes used, are equipped with an additional PCI-E X4 100GB SSD. The operating system is Ubuntu 9.04, the parallel file system installed is OrangeFS v2.8.8 and the KVStore storage system is Hyperdex v1.3. All nodes are equipped with Gigabit Ethernet interconnection.

We should point out that our choice of those specific storage systems as representatives from each category e.g file-based and KVstore-based was made due to various reasons. First of all, OrangeFS is a widely known and used PFS in the HPC community and it is mature enough in terms of development and research as to be the representative for the file-based storage system. As fas as Hyperdex, this is a relatively new implementation by Cornell University, it is open source and it has somehow easy APIs. It is well documented and it has active support by its developers. We believe that this implementation is as good as any other KVStore solution and as its designers claim it is 12-13x faster than Cassandra and MongoDB which are considered quite established up to date.

For each set of testing cases, in OrangeFS we used 4 nodes both as storage nodes and as metadata nodes. For HyperDex, we also used 4 nodes as storage nodes and a separate node as the coordinator node (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 since KVStore involves lighter metadata operations and has a totally different architecture on metadata management, after careful consideration we have concluded that this setup is fair and would not be any kind of a bottleneck for any of those systems.

A. Methodology

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

1) Tracing: IOSIG [8], an I/O pattern analysis tool developed in I/O middle-ware level, is used to capture the run-time statistics of data accesses. Using these information, we were able to identify the key characteristics of the I/O behaviour of the application. The IOSIG trace includes a lot of information such as Process ID, Offset, Request size, Begin time, End time and others. We only considered the offset and the request size since these two values alone, can determine the access pattern of the application.

2) Trace Player / Workload Generator: Having the desired information extracted from the trace, we designed and implemented a fairly straightforward workload generator. This workload generator takes an I/O trace as input and it "replays" all the I/O operations onto the file system that is being tested. Practically, we developed three trace players, one for OrangeFS, one for HyperDex and a third one for OrangeFS again but modified to simulate some extra metadata operations. The reason we designed this last one is mostly 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, do the I/O on this file and then closes it. The new trace player for OrangeFS, for each request to the file system for I/O, it opens a small file, do the I/O and then closes the file but it's doing this for every request found in the trace. This way, the amount of metadata produced by OrangeFS are similar to that of HyperDex. This new workload generator is mostly simulating the behaviour of OrangeFS when operating with many small files in applications such as graph applications. It is 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 a similar case. With these three workload generators, it is easy to test the systems under various workloads since the only thing needed is to feed the I/O trace into the appropriate trace player and measure the time spent on I/O operations.

3) Performance Measurement: To measure the performance of each storage system, we wrapped each I/O operation under a time barrier and calculate the total time spent for I/O. When doing so, special attention needs to be taken so total time doesn't include other operations such as system start up or other preparations before the actual I/O operation. In order to

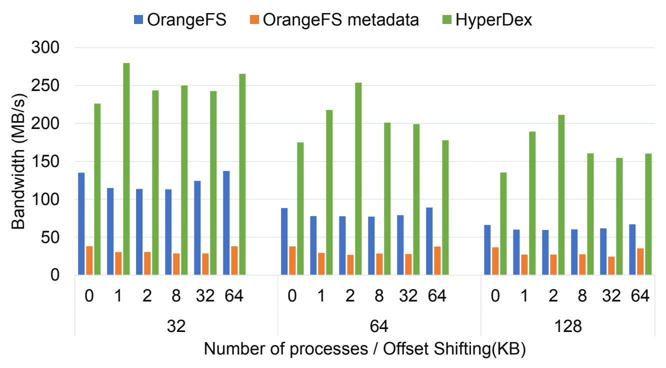


Fig. 8. Shifting offset comparison.

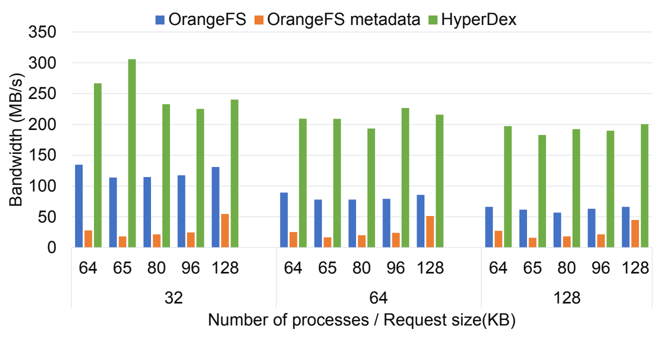


Fig. 9. Varying request size comparison.

focus on the file system performance, we removed the effects of memory cache and buffer. Before each run of the test, we clean the operating system cache to ensure all data will be read from the storage devices. Prior to the first run, we also prepared the data for both systems. The experiments are measuring the read performance and so the data were already there. 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 key and the request size as value. We also run 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, overloaded network, etc.

B. Results with Synthetic Benchmarks

We wanted to test specific access patterns that seem to affect the performance on a traditional parallel file system like 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.

1) Offset Shifting: The first synthetic benchmark was designed to simulate an unaligned with the stripes of the parallel file system access pattern where the offset of the next request is shifted by some bytes and thus forcing the system to coordinate each sub-request amongst multiple storage nodes. This specific access pattern clearly stresses OrangeFS but it doesn't seem to be a problem for the KVStore system where there is no need to synchronize the sub-requests since each request is for a different object.

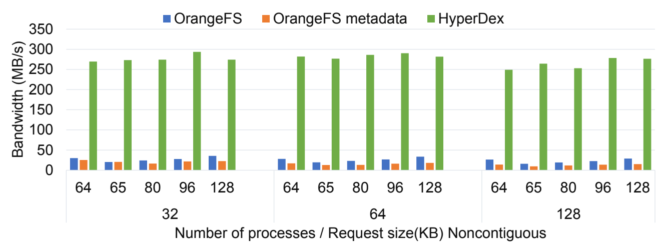
Figure 8 shows the comparison between OrangeFS and HyperDex for this offset shifting testing case. We captured the I/O trace of this synthetic trace and then gave it as input to the workload generator for each system. Additionally, we run it with the modified OrangeFS benchmark with increased metadata 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.

2) Varying Request Size: The second synthetic benchmark is to simulate a varying request size access pattern. The default value of the stripe size in OrangeFS is 64KB. When a request is 64KB or a multiply of that value, then it is well aligned with the stripes on each node. Generally, in parallel file systems, stripe sizes are fixed. Its hard to match them with various request sizes. If a request is not aligned with the striping pattern such decomposition can make the first and last sub-requests much smaller than the striping unit. This can lead to some serious degradation, as we showed in Section II. This benchmark is stressing the storage system exactly on that access pattern. We tried some different cases where each process is issuing requests with 64KB, 65KB, 80KB, 96KB, and 128KB size for each case and the results can be seen in Figure 9. HyperDex clearly isn't affected again and the performance is higher from 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 by this access pattern and also by the amount of the metadata operations. HyperDex achieved a higher bandwidth by an average of 914%. Specifically, this benchmark run on HDDs and HyperDex performed at around 200 MB/s where OrangeFS with increased metadata operations was around 35 MB/s.

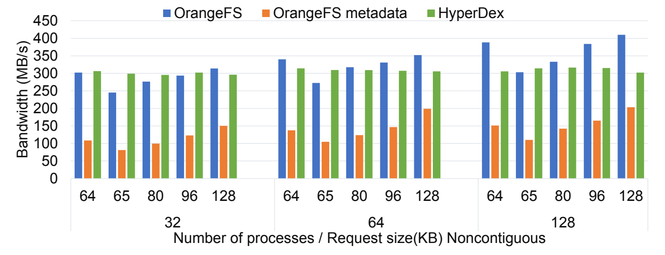
3) Noncontiguous Access Pattern: In this last synthetic benchmark, a noncontiguous access pattern is produced where a gap between each request is created and thus forcing the storage system to move across the file to do the requested I/O operation. Basically, each request is of various size and is served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

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

But, if we look the SSD case, the picture is quite different. OrangeFS demonstrated a very good bandwidth and in



(a) HDD



(b) SSD

Fig. 10. Noncontiguous access pattern comparison.

some cases it surpasses HyperDex. There is some obvious degradation for OrangeFS where for 64KB request size at 128 number of processes the bandwidth was close to 400 MB/s where fro 65KB request size on the other hand was 300 MB/s making a 24% degradation. Even with this degradation though, OrangeFS managed to keep the performance very high and compared to HyperDex, which was very stable, it had a 106% in average performance gain. Finally, OrangeFS with increased metadata operations, demonstrated a relatively low performance compared with both the normal OrangeFS and HyperDex as well. HyperDex performed higher by an average of 238% compared with the OrangeFS with increased metadata operations.

C. Results with HPIO Benchmark

The HPIO (High-performance I/O) benchmark is a tool for evaluating/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 and it was designed and implemented by Northwestern University. We designed a noncontiguous access pattern with a fixed request size of 64KB and measured the performance of these two storage systems.

Figure 11 demonstrates the comparison between OrangeFS and HyperDex. Remarkably, HyperDex was very stable and it achieved a bandwidth of about 240 MB/s. OrangeFS seemed to suffer from this particular access pattern and as the number of processes increased, the performance was decreased and reached a low of 63 MB/s. Comparing these two, HyperDex offered a better performance by an average 241%. OrangeFS with increased metadata performed even lower where the bandwidth was merely 50 MB/s.

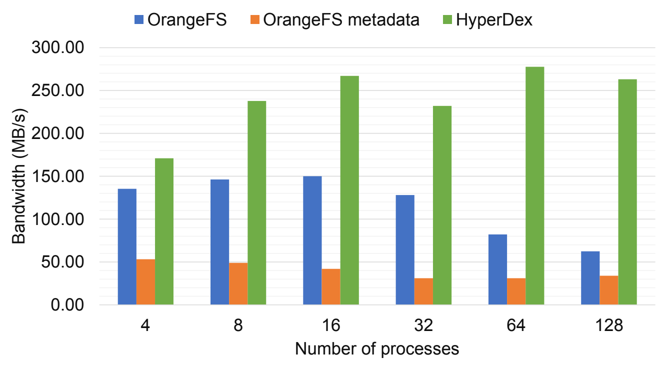


Fig. 11. HPIO trace comparison.

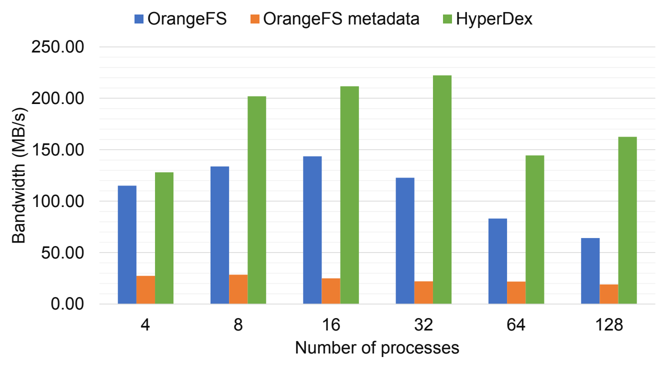


Fig. 12. LANLApp1 trace comparison.

D. Results with a Real Application

To take this comparison even further to real world scientific applications, we took the I/O trace of LANL Anonymous App1 and feed it to the workload generator. In this application, there are three I/O requests in each loop, one small request with 16 bytes followed by two large requests with (128K-16) bytes and 128 KB respectively.

In Figure 12 we can observe that HyperDex hit a bandwidth of 220 Mb/s in the case of 32 processes whereas OrangeFS was only at 120 MB/s. In average, HyperDex achieved a 179 MB/s bandwidth and OrangeFS a 110 MB/s. When the increased metadata scenario run, OrangeFS performed really bad and it had a 25 MB/s average bandwidth resulting in an impressive 756% difference with HyperDex.

V. Related Work

Parallel file systems are the de-facto method of data storage for HPC systems. They provide high data access performance and the consistent file based storage space. Popular PFSs include Lustre [9], OrangeFS, GPFS [10], etc. Researchers found that the server-side data layout and client-side data access pattern can largely affect the overall I/O performance, because they affects the mapping between the logical data requests from the applications and the physical data layout on server nodes [11]. The parallel data access between multiple client nodes and multiple server nodes inevitably brings data synchronization. Song [2] designed an I/O coordination scheme to reduce the average completion time for concurrent applications. Zhang [12] also noticed the sub-request data synchronization caused performance degradation, and 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 Zhang's iBridge work.

Most parallel file systems support the POSIX standard, however in many cases POSIX is unnecessarily strict [13]. It may heart the system's scalability and the ability to control the small objects contained in a file independently [14]. Object based systems and Key-Value Stores provides better flexibility with an object based interface, instead of the file based interface. KVStore is widely used in internet services and cloud storage services [15], but it is rarely used for HPC systems. Many existing works have compared the advantages and disadvantages of file systems and object storage systems [16] [15] [17]. Some other works tried to integrate PFS and object storage system [18] or expose the underlying object data streams of PFS [14] to gain better I/O performance. This study tries to explore whether HPC workloads can benefit from object based system like KVStore.

VI. 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 ran 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.

VII. Conclusion

PFSs are the dominant storage systems in HPC systems. KVStores are widely used by Internet and Cloud storage service, but are rarely used by HPC systems. By thoughtfully examining the performance characteristics of PFS and KVStore, we proposal to utilize KVStore to optimize some workload's I/O performance, especially for the workloads that does not favor PFS.

We conducted comprehensive and extensive experiments and the results proved the value of our proposal. It is noteworthy that, our proposal is not to replace PFS with KVStore. A PFS's performance can be very high for its ideal workloads, and it can also be very low for some irregular workloads. With the same hardware, KVStore's performance is stable somewhere between PFS's high and low points. So, it is valuable to optimize the performance with KVStore for PFS's low-performance cases.

In our future work, we want to extent the experimentation in larger scales and with better network interfaces like InfiniBand. We may explore the comparison between different representatives from PFSs and KVStores. Finally, we also plan to build a performance model to help users decide what storage system (PFS or KVStore) they should choose for their target workloads.

References

[1] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2009.

[2] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-Side I/O Coordination for Parallel File Systems," in *Proceedings of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, 2011.

[3] (2014, August) Orange File System. [Online]. Available: http://www.orangefs.org

[4] P. H. Carns, W. B. L. III, R. B. Ross, R. Thakur *et al.*, "PVFS: A parallel file system for Linux clusters," in *in Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[5] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A Distributed, Searchable Key-Value Store," in *Proceedings of the SIGCOMM Conference*, 2012.

[6] A. Ching. (2014, Aug.) HPIO I/O Benchmark. [Online]. Available: http://goo.gl/OEaaiB

[7] (2014, Aug.) PLFS I/O Traces. [Online]. Available: http://institutes.lanl.gov/plfs/maps

[8] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting Application-Specific Parallel I/O Optimization Using IOSIG," in *Proceedings of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2012.

[9] P. J. Braam *et al.* (2014, Mar.) The Lustre storage architecture. [Online]. Available: ftp://ftp.uni-duisburg.de/linux/filesys/Lustre/lustre.pdf

[10] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of USENIX Conference on File and Storage Technologies*, 2002.

[11] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A Cost-Intelligent Application-Specific Data Layout Scheme for Parallel File Systems," in *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, 2011.

[12] X. Zhang, K. Liu, K. Davis, and S. Jiang, "iBridge: Improving unaligned parallel file access with solid-state drives," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2013.

[13] D. Kimpe and R. Ross, "Storage Models: Past, Present, and Future," in *High Performance Parallel I/O*, 2014, ch. 30.

[14] D. Goodell, S. J. Kim, R. Latham, M. Kandemir, and R. Ross, "An Evolutionary Path to Object Storage Access," in *Proceedings of the Parallel Data Storage Workshop*, 2012.

[15] "EMC Object-based Storage for Active Archiving and Application Development," The TANEJA Group, Inc., Tech. Rep., 2012. [Online]. Available: http://www.emc.com/collateral/analyst-reports/emc-atmosecosystem-taneja-group-tech-brief-final-ar.pdf

[16] M. J. Brim, D. A. Dillow, S. Oral, B. W. Settlemyer, and F. Wang, "Asynchronous object storage with QoS for scientific and commercial big data," in *Proceedings of the Parallel Data Storage Workshop*, 2013.

[17] G. A. Gibson, J. S. Vitter, and J. Wilkes, "Strategic directions in storage I/O issues in large-scale computing," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 779–793, 1996.

[18] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan, "Integrating Parallel File Systems with Object-based Storage Devices," in *Proceedings of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, 2007.