FusionFS: Toward Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems

Dongfang Zhao[⋆◇], Zhao Zhang[†], Xiaobing Zhou[‡], Tonglin Li[⋆], Ke Wang[⋆],
Dries Kimpe[◇], Philip Carns[◇], Robert Ross[◇], and Ioan Raicu[⋆◇]

[⋆]Illinois Institute of Technology [†]UC Berkeley [‡]Hortonworks Inc. [◇]Argonne National Laboratory

dzhao8@iit.edu, zhaozhang@eecs.berkeley.edu, xzhou@hortonworks.com, {tli33, kwang22}@hawk.iit.edu,
{dkimpe, carns, rross}@mcs.anl.gov, iraicu@cs.iit.edu

***Abstract*—State-of-the-art, yet decades-old, architecture of high-performance computing systems has its compute and storage resources separated. It thus is limited for modern data-intensive scientific applications because every I/O needs to be transferred via the network between the compute and storage resources. In this paper we propose an architecture that hss a distributed storage layer local to the compute nodes. This layer is responsible for most of the I/O operations and saves extreme amounts of data movement between compute and storage resources. We have designed and implemented a system prototype of this architecture—which we call the FusionFS distributed file system—to support metadata-intensive and write-intensive operations, both of which are critical to the I/O performance of scientific applications. FusionFS has been deployed and evaluated on up to 16K compute nodes of an IBM Blue Gene/P supercomputer, showing more than an order of magnitude performance improvement over other popular file systems such as GPFS, PVFS, and HDFS.**

I. Introduction

The conventional architecture of high-performance computing (HPC) systems separates the compute and storage resources into two parts, compute nodes and storage nodes, which are interconnected by a shared network infrastructure. This architecture results mainly from the nature of many legacy large-scale scientific applications that are compute intensive, where it is often assumed that the storage I/O capabilities are lightly utilized for the initial data input, some periodic checkpoints, and the final output. In the era of big data, however, scientific applications such as astronomy [1] are becoming more and more data-intensive, requiring a greater degree of support from the storage subsystem [2]. Our previous simulation work [3, 4] demonstrated that the current HPC storage architecture would not scale to the emerging exascale computing systems (10^{18} ops/s).

While recent studies [5, 6] have addressed the I/O bottleneck in the conventional architecture of HPC systems, the work presented here is orthogonal to those studies in that we propose a new storage architecture that colocates the storage and compute resources. The ideas in this paper build on prior work [7] presented at the 2012 Supercomputing conference as a poster. In particular, we envision a distributed storage system on compute nodes enabling applications to manipulate their intermediate results and checkpoints, rather than transferring data over the network. Colocation of storage and computation has been widely leveraged in data centers (e.g., Hadoop clusters). Nevertheless, such an approach has not been implemented in HPC systems despite its having attracted considerable research interest recently; see, for example, the DEEP-ER [8] project funded by the European Union. We demonstrate here how to architect and engineer such a system, and we report how much, quantitatively, it could improve the I/O performance of real-world scientific applications.

Arguably, colocating compute and storage could raise concerns about jitters on compute nodes, since an application's computation and I/O share resources such as CPU and network. We argue that the I/O-related cost can be offloaded onto dedicated infrastructures that are decoupled from the application's acquired resources, as justified in [9]. In fact, this resource-isolation strategy has been applied in production systems: the IBM Blue Gene/Q supercomputer (Mira [10]), for example, assigns one core of the chip (17 cores in total) for the local operating system and the other 16 cores for applications.

Distributed storage has been extensively studied in data centers (e.g., the popular distributed file system HDFS [11]); yet little literature exists on building a distributed storage system particularly for HPC systems, whose design principles are much different from data centers. HPC nodes are highly customized and tightly coupled with a high-throughput and low-latency network (e.g., InfiniBand), whereas data centers typically have commodity servers and inexpensive networks (e.g., Ethernet). Therefore, storage systems designed for data centers are not optimized for the HPC machines; indeed, as we will discuss in more detail in Section V, HDFS shows poor performance on a typical HPC machine. In particular, we observe that the following challenges are unique to a distributed file system on HPC compute nodes, related to both metadata-intensive and write-intensive workloads.

First, the storage system on HPC compute nodes needs to support intensive metadata operations. Many scientific applications create a large number of small- to medium-sized files: for example, Welch and Noer [12] reported that 25%–90% of all the 600 million files from 65 Panasas [13] installations are 64 KB or smaller. Thus the I/O performance is highly throttled by the metadata rate, besides the data itself. Data centers, however, are not optimized for this type of workload. Recall that HDFS [11] splits a large file into a series of default 64 MB chunks (128 MB recommended in most cases) for parallel processing. Thus, a small- or medium-sized file can benefit little from this data parallelism. Moreover, the centralized metadata server in HDFS is apparently not designed to handle intensive metadata operations.

Second, file writes should be optimized for a distributed file system on HPC compute nodes. The fault tolerance of most of today's large-scale HPC systems is achieved through some form of checkpointing. In essence, the system periodically flushes memory to external persistent storage and occasionally loads the data back to memory in order to roll back to the most recent correct checkpoint after a failure. Hence, file writes typically outnumber file reads in terms of both frequency and size in HPC systems, and improving the write performance will significantly reduce the overall I/O cost. The fault tolerance of data centers, however, is achieved not through checkpointing its memory states but through recomputing affected data chunks that are replicated on multiple nodes.

To overcome these challenges, we have designed and implemented FusionFS to disperse its metadata to all the available compute nodes, in order to achieve the maximal concurrency of metadata operations. Every client of FusionFS optimizes write operations with local writes (whenever possible), an approach that reduces network traffic and makes the aggregate I/O throughput highly scalable. We expect FusionFS to coexist with the remote parallel file system (e.g., GPFS [14]) rather than to replace the latter, because the compute nodes of current HPC systems are tightly coupled and are not viable to provide on-board storage as large as the remote parallel file systems.

FusionFS has been deployed on up to 16K compute nodes of an IBM Blue Gene/P supercomputer (Intrepid [15]) and heavily accessed by a variety of benchmarks and applications. We observed more than an order of magnitude improvement to the I/O performance when using FusionFS compared with other popular file systems such as GPFS [14], PVFS [16], and HDFS [11], surpassing 2.5 TB/s aggregate I/O throughput on 16K nodes. In addition, FusionFS has been serving as the infrastructure or test bed of a few related projects such as virtual-chunk-based file compression [17, 18].

In summary, this paper makes the following contributions:

- *Proposal of an unprecedented storage architecture for extreme-scale HPC systems to address the I/O bottleneck of modern data-intensive scientific applications*
- *Design and implementation of the FusionFS filesystem to support metadata- and write-intensive workloads*
- *Evaluation of FusionFS with benchmarks and applications at extreme scales, and demonstration of its superiority over state-of-the-art solutions*

II. Design Overview

As shown in Figure 1, FusionFS is a user-level file system that runs on the compute resource infrastructure and enables every compute node to actively participate in both the metadata and data movement. The client (or application) is able to access the global namespace of the file system with a distributed metadata service. Metadata and data are completely decoupled: the metadata on a particular compute node does not necessarily describe the data residing on the same compute node. The decoupling of metadata and data allows different strategies to be applied to metadata and data management, respectively.

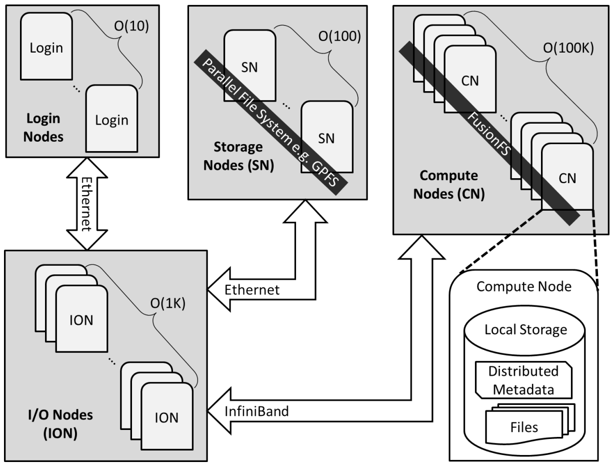


Figure 1. FusionFS deployment in a typical HPC system

FusionFS supports both the POSIX interface and a user library. The POSIX interface is implemented with the FUSE framework [19], so that legacy applications can run directly on FusionFS without modifications. Just like other user-level file systems (e.g., PVFS [16]), FusionFS can be deployed as a mount point in a UNIX-like system. The mount point is a virtual root directory to the clients when using FusionFS.

Users need to specify three arguments when deploying FusionFS as a POSIX-compliant mount point on a compute node: the scratch directory in which to store the metadata and data, the mount point of the remote parallel file system (e.g., Lustrue [20], GPFS [14], PVFS [16]), and the mount point of FusionFS in which applications manipulate files. The remote parallel file system needs to be integral to the global namespace because it is necessary to accommodate large files that cannot fit in FusionFS.

FUSE has been criticized for its performance overhead. Native UNIX-like file systems (e.g., Ext4) have only two context switches between the user space and the kernel. In contrast, for a FUSE-based file system, context needs to be switched four times: two switches between the caller and VFS and another two between the FUSE user library (*libfuse*) and the FUSE kernel module (*/dev/fuse*). A detailed comparison between FUSE-enabled and native file systems [21] shows that a Java implementation of a FUSE-based file system introduces about 60% overhead compared with the native file system. However, in the context of C/C++ implementation with multithreading on memory-level storage, which is a typical setup in HPC systems, the overhead is much lower. In prior work [22], we reported that FUSE could deliver as high as 578 MB/s throughput, 85% of the raw bandwidth.

To avoid the performance overhead from FUSE, FusionFS also provides a user library for applications to directly interact with their files. These APIs look similar to those of POSIX, for example *ffs_open()*, *ffs_close()*, *ffs_read()*, and *ffs_write()*. The downside of this approach is the lack of POSIX support, indicating that the application might not be portable to other file systems and often needs some modifications and recompilation.

III. Metadata Management

In this section we discuss the design and implementation of the distributed metadata management in FusionFS.

A. Namespace

Clients have a coherent view of all the files in FusionFS, whether the file is stored in the local node or in a remote node. This global namespace is maintained by a distributed hash table (DHT [23, 24]), which disperses partial metadata on each compute node and has served as the infrastructure for a few other systems such as data provenance [25, 26] and key-value stores [27]. As shown in Figure 2, in this example Node 1 and Node 2 physically store only two subgraphs (the top left and top right portion of the figure) of the entire metadata graph. The client could interact with the DHT to inquire any file on any node, as shown in the bottom portion of the figure. Because the global namespace is just a logical view for clients and does not physically exist in any data structure, the global namespace need not need be aggregated or flushed when changes occur to the subgraph on local compute nodes. The changes to the local metadata storage will be exposed to the global namespace when the client queries the DHT.

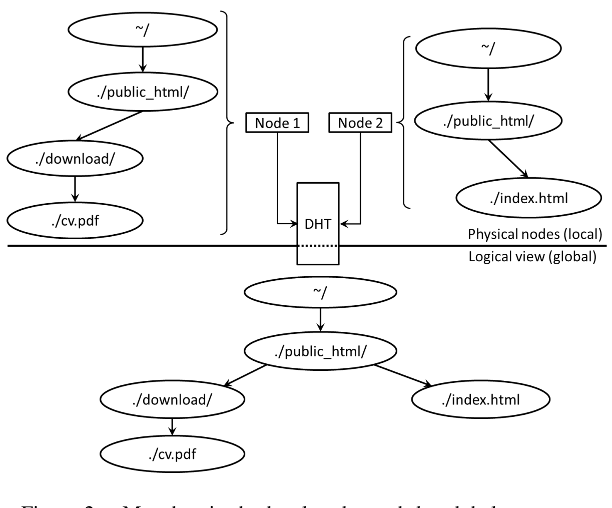


Figure 2. Metadata in the local nodes and the global namespace

B. Data Structures

FusionFS has different data structures for managing regular files and directories. For a regular file, the field *addr* stores the node where this file resides. For a directory, the field *filelist* records all the entries under this directory. This *filelist* field is particularly useful for providing in-memory speed for directory read, for example, "ls /mnt/fusionfs." Nevertheless, both regular files and directories share some common fields, such as timestamps and permissions, which are commonly found in traditional i-nodes.

To make matters more concrete, we show in Figure 3 the distributed hash table according to the example metadata shown in Figure 2. Here, the DHT is only a logical view of the aggregation of multiple partial metadata on local nodes (in this case, Node 1 and Node 2). Five entries (three directories, two regular files) are stored in the DHT, with their file names as keys. The value is a list of properties delimited by semicolons. For example, the first and second portions of the values are permission flag and file size, respectively. The third portion for a directory value is a list of its entries delimited by commas, while for regular files it is just the physical location of the file, for example, the IP address of the node on which the file is stored. Upon a client request, this value structure is serialized by Google Protocol Buffers [28] before sending over the network to the metadata server, which is just another compute node. Similarly, when the metadata blob is received by a node, we deserialize the blob back into the C structure with Google Protocol Buffers.

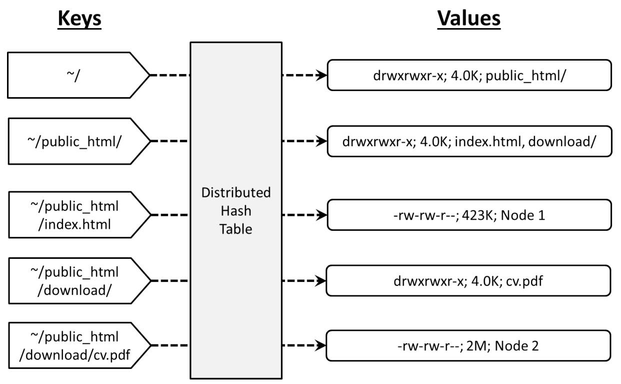


Figure 3. Global namespace abstracted by key-value pairs in a DHT

The metadata and data on a local node are completely decoupled: a regular file's location is independent of its metadata location. This flexibility allows us to apply different strategies to metadata and data management, respectively. Moreover, the separation between metadata and data has the potential to plug in alternative components to metadata or data management, making the system more modular.

As Figure 2 shows, the $index.html$ metadata is stored on Node 2, and the $cv.pdf$ metadata is on Node 1. However, $index.html$ can reside on Node 1, and $cv.pdf$ can reside on Node 2, as shown in Figure 3. Besides the conventional metadata information for regular files, the value has a special flag indicating whether this file is being written. Any client who requests to write a file needs to set this flag before opening the file; the client will not reset it until the file is closed. The atomic compare-swap operation supported by DHT [23] guarantees the consistency for concurrent writes.

Another challenge facing the metadata implementation is large-directory performance issues. In particular, when a large number of clients write many small files on the same directory concurrently, the value of this directory in the key-value pair gets incredibly long and responds extremely slowly. The reason is that a client needs to update the entire old long string with the new one, even though the majority of the old string is unchanged. To fix that issue, we implemented an atomic append operation that asynchronously appends the incremental change to the value. This approach is similar to Google File System [29], where files are immutable and can only be appended. This gives us excellent concurrent metadata modification in large directories, at the expense of potentially slower directory metadata read operations.

C. Network Protocols

We encapsulate several network protocols in an abstraction layer. Users can specify which protocol to be applied in their deployments. Currently, we support three protocols: TCP, UDP, and MPI. Since we expect a high network concurrency on metadata servers, epoll [30] is used instead of multithreading. The side effect of epoll is that the received message packets are not kept in the same order as on the sender. To address this, we added a header [message_id, packet_id] to the message at the sender, and the message is restored by sorting the packet_id for each message at the recipient. This procedure is done efficiently by a sorted map with message_id as the key, mapping to a sorted set of the message's packets.

D. Persistence

The objective of the proposed distributed metadata architecture is to improve performance. Thus, any metadata manipulation from clients should occur in memory, plus some network transfer if needed. On the other hand, persistence is required for metadata in case of any memory errors or system restarts.

The persistence of metadata is achieved by periodically flushing the in-memory metadata onto the local persistent storage. In some sense, the process is similar to the incremental checkpointing mechanism. This asynchronous flushing helps sustain the high performance of the in-memory metadata operations.

E. Consistency

Since each primary metadata copy has replicas, the next question is how to make them consistent. Traditionally, two semantics are used to keep replicas consistent: (1) strong consistency – blocking until replicas are finished with updating and (2) weak consistency – returning immediately when the primary copy is updated. The tradeoff between performance and consistency is tricky, most likely depending on the workload characteristics.

As for a system design without any *a priori* information on the particular workload, we compromise with both sides: assuming the replicas are ordered by some criteria (e.g., last modification time), the first replica shows strong consistency with the primary copy, and the other replicas are updated asynchronously. With this approach, the metadata shows strong consistency (in the average case) while the overhead is kept relatively low.

IV. File Manipulation

In this section we discuss the protocols for manipulating files in FusionFS.

A. Network Transfer

For file transfer, neither UDP nor TCP is ideal for FusionFS on HPC compute nodes. UDP is a highly efficient protocol but lacks reliability support. TCP, on the other hand, supports reliable transfer of packets but adds significant overhead.

We therefore have developed our own data transfer service Fusion Data Transfer (FDT) on top of UDP-based Data Transfer (UDT) [31]. UDT is a reliable UDP-based application-level data transport protocol for distributed data-intensive applications. UDT adds its own reliability and congestion control on top of UDP that offers a higher speed than does TCP.

B. File Open

Figure 4 shows the protocol when opening a file in FusionFS. Because of limited space, we assume here that the requested file is also on Node-j. Note that it is not necessarily Node-j that stores both the requested file and its metadata; as we explained in Section III-B, the metadata and data are decoupled on compute nodes.

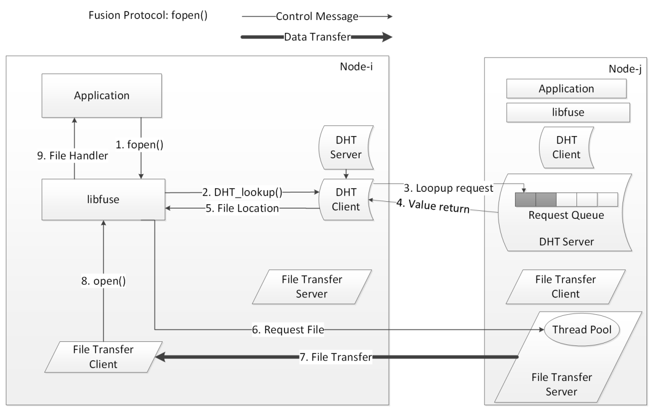


Figure 4. Protocol of file open in FusionFS

In step 1, the application on Node-i issues a POSIX fopen() call that is caught by the implementation in the FUSE user-level interface (i.e., *libfuse*) for file open. Steps 2–5 retrieve the file location from the metadata service that is implemented by a distributed hash table [23]. The location information might be stored in another machine Node-j, so this procedure could involve a round trip of messages between Node-i and Node-j. Then Node-i needs to ping Node-j to fetch the file in steps 6–7. Step 8 triggers the system call to open the transferred file. Step 9 returns the file handle to the application.

C. File Write

Before writing to a file, the process checks whether the file is being accessed by another process, as discussed in Section III-B. If so, an error number is returned to the caller. Otherwise the process can do one of the following: (1) if the file is originally stored on a remote node, the file is transferred to the local node in the *fopen()* procedure, after which the process writes to the local copy; or (2) if the file to be written is right on the local node, or it is a new file, then the process starts writing the file just like a system call.

The aggregate write throughput is optimal because file writes are associated with local I/O throughput and avoid the following two types of cost: (1) determining to which node the data will be written, normally accomplished by pinging the metadata nodes or some monitoring services, and (2) transferring the data to a remote node. The downside of this file write strategy is the poor control on the load balance of compute node storage. This issue could be addressed by an asynchronous rebalance procedure running in the background or by a load-aware task scheduler that steals tasks from the active nodes to the more idle ones.

When the process finishes writing to a file that is originally stored in another node, FusionFS does not send the newly modified file back to its original node. Instead, the metadata of this file is updated. This approach saves the cost of transferring the file data over the network.

D. File Read

Unlike file write, it is impossible to arbitrarily control where the requested data reside for file read. The location of the requested data is highly dependent on the I/O pattern. However, we could determine which node the job is executed on by the distributed workflow system, for example, Swift [32, 33]. That is, when a job on node A needs to read some data on node B, we reschedule the job on node B. The overhead of rescheduling the job is typically smaller than transferring the data over the network, especially for data-intensive applications. In our previous work [34], we detailed this approach and justified it with theoretical analysis and experiments on benchmarks and real applications.

Indeed, remote readings are not always avoidable for some I/O patterns. For example, in merge sort, the data needs to be joined together, and shifting the job cannot avoid the aggregation. In such cases, we need to transfer the requested data from the remote node to the requesting node. The data movement across compute nodes within FusionFS is conducted by the FDT service discussed in Section IV-A. FDT service is deployed on each compute node and keeps listening to the incoming fetch and send requests.

E. File Close

Figure 5 shows the protocol when closing a file in FusionFS. In steps 1–3 the application on Node-i closes and flushes the file to the local disk. If this is a read-only operation before the file is closed, then *libfuse* only needs to signal the caller (i.e., the application) in step 10. If this file has been modified, then its metadata needs to be updated in steps 4–7. Moreover, the replicas of this file also need to be updated in steps 8–9.

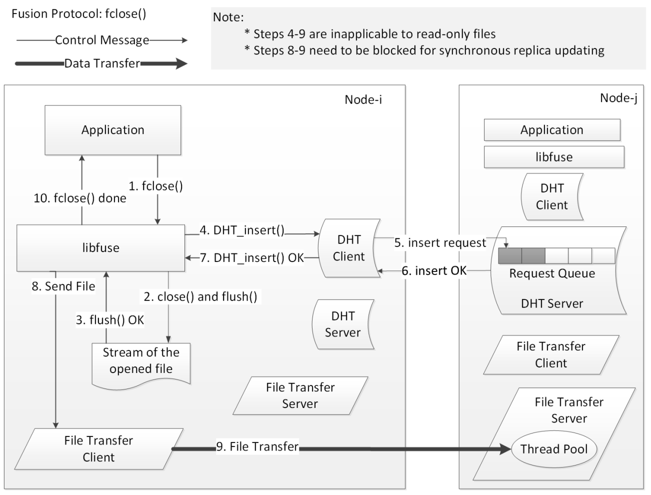


Figure 5. Protocol of file close in FusionFS

Again, as in Figure 4, the replica is not necessarily stored on the same node of its metadata (Node-j). To save space here, we show just its remote replica on Node-j.

V. Evaluation

While we indeed compare FusionFS with some open-source systems such as PVFS [16] (in Figure 7) and HDFS [11] (in Figure 11), our top mission is to evaluate its performance improvement over the production file system of today's fastest systems. If we look at today's top 10 supercomputers [35], 4 systems are IBM Blue Gene/Q systems that run GPFS [14] as the default file system. Therefore, most large-scale experiments reported in this paper were carried out on Intrepid [15], a 40K-node IBM Blue Gene/P supercomputer whose default file system is also GPFS. Each Intrepid compute node has quad core 850 MHz PowerPC 450 processors and runs a lightweight Linux ZeptoOS [36] with 2 GB memory. A 7.6 PB GPFS [14] parallel file system is deployed on 128 storage nodes. When FusionFS is evaluated as a POSIX-compliant file system, each compute node gets access to a local storage mount point with 174 MB/s throughput on par with today's high-end hard drives. It points to the ramdisk and is throttled by a single-threaded FUSE layer. The network protocols for metadata management and file manipulation are TCP and FDT, respectively.

All experiments were repeated at least five times until results became stable (within 5% margin of error). The reported numbers are the average of all runs. Caching effects are carefully precluded by reading a file larger than the on-board memory before the measurement.

A. Metadata Rate

We expect that the metadata performance of FusionFS should be significantly higher than that of the remote GPFS on Intrepid because FusionFS manipulates metadata in a completely distributed manner on compute nodes whereas GPFS has a limited number of clients on I/O nodes (every 64 compute nodes share one I/O node in GPFS). To quantitatively study the improvement, we have both FusionFS and GPFS create 10K empty files from each client on its own directory on Intrepid. That is, at the 1024-node scale, we create 10M files over 1,024 directories. We could have let all clients write on the same directory, but this workload would not take advantage of the multiple I/O nodes of GPFS. That is, we want to optimize GPFS's performance when comparing it with that of FusionFS.

As shown in Figure 6, at the 1024-node scale, FusionFS delivers a metadata rate nearly two orders of magnitude higher than that of GPFS. FusionFS shows excellent scalability, with no sign of slowdown up to 1,024 nodes. The gap between GPFS and FusionFS metadata performance would continue to grow, since eight nodes are enough to saturate the metadata servers of GPFS.

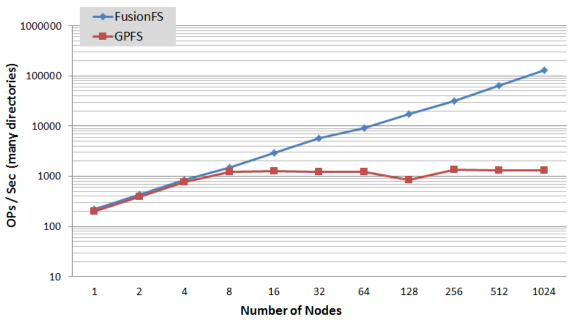


Figure 6. Metadata performance of FusionFS and GPFS on Intrepid (many directories)

One might overlook FusionFS's novel metadata design and state that GPFS is slower than FusionFS simply because GPFS has fewer metadata servers (128) and fewer I/O nodes (1:64). We emphasize, however, that FusionFS was designed for this very purpose: to maximize the metadata concurrency without adding new resources to the system.

To really answer the question "What if a parallel file system has the same number of metadata servers as does FusionFS?", we installed PVFS [16] on Intrepid compute nodes with the 1-1-1 mapping between clients, metadata servers, and data servers, just as it is in FusionFS. We did not deploy GPFS on compute nodes because it is a proprietary system and PVFS is open source. The results are shown in Figure 7. Both FusionFS and PVFS turn on the POSIX interface with FUSE. Each client creates 10K empty files on the same directory to push more pressure on both systems' metadata service. FusionFS outperforms PVFS even for a single client, thus illustrating that the metadata optimization for the big directory (i.e., update $\rightarrow$ append) on FusionFS is highly effective. Moreover, FusionFS again shows linear scalability. On the other hand, PVFS is saturated at 32 nodes, suggesting that more metadata servers in parallel file systems do not necessarily improve the ability to handle higher concurrency.

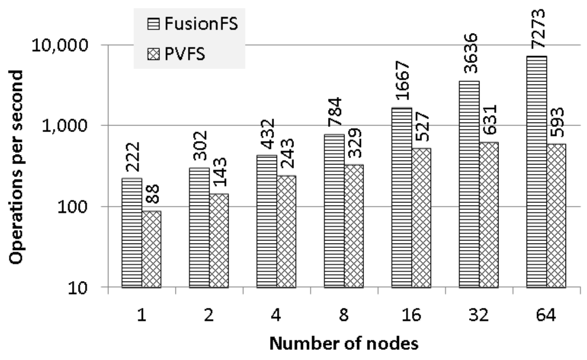


Figure 7. Metadata performance of FusionFS and PVFS on Intrepid (single directory)

B. I/O Throughput

As with the metadata, we expect a significant improvement in the I/O throughput from FusionFS. Figure 8 shows the aggregate write throughput of FusionFS and GPFS on up to 1,024 nodes of Intrepid. FusionFS shows almost linear scalability across all scales. GPFS scales at a 64-node step because every 64 compute nodes share one I/O node. Nevertheless, GPFS is still orders of magnitude slower than FusionFS at all scales.

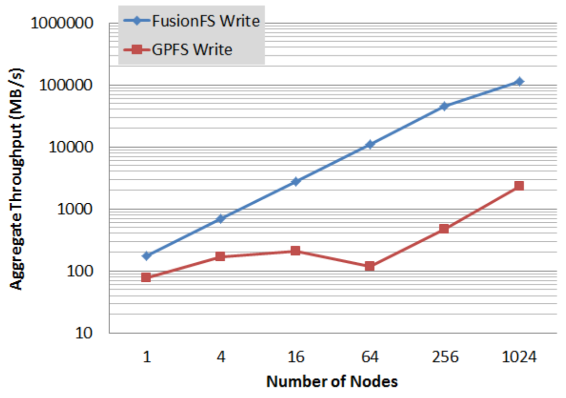


Figure 8. Write throughput of FusionFS and GPFS on Intrepid

Figure 9 shows the scalability of FusionFS at extreme scales. The experiment was carried out on Intrepid on up to 16K nodes each of which has a FusionFS mount point. FusionFS throughput shows almost linear scalability: doubling the number of nodes yields doubled throughput. Specifically, we observe stable 2.5 TB/s throughput (peak 2.64 TB/s) on 16K nodes.

The main reason that FusionFS data write is faster is that the compute node writes only to its local storage. This is not true for data read, however: it is possible that one node needs

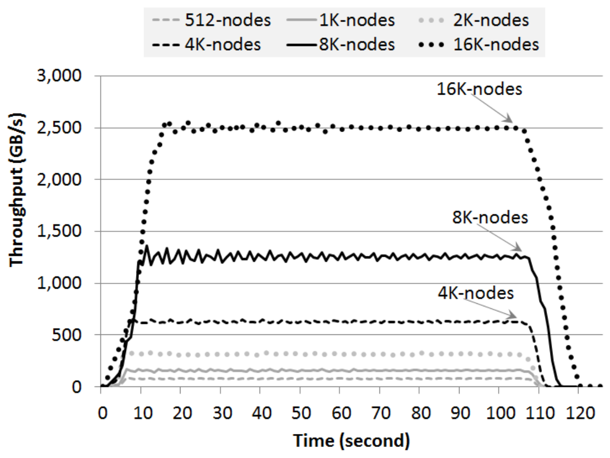


Figure 9. FusionFS scalability on Intrepid

to transfer some remote data to its local disk. Thus, we are interested in two extreme scenarios—all-local read and all-remote read—that define the lower and upper bounds of read throughput. We measured FusionFS for both cases on 256 nodes of Intrepid, where each compute node reads a file of different sizes from 1 MB to 256 MB. For the all-local case (e.g., where a data-aware scheduler can schedule tasks close to the data), all the files are read from the local nodes. For the all-remote case (e.g., where the scheduler is unaware of the data locality), every file is read from the next node in a round-robin fashion. This I/O pattern is unlikely in real-world applications but serves well as a workload for an all-remote request.

Figure 10 shows that FusionFS all-local read outperforms GPFS by more than one order of magnitude, as we have seen for data write. The all-remote read throughput of FusionFS is also significantly higher than GPFS, even though not as considerable as the all-local case. The reason all-remote reads still outperform GPFS is, again, FusionFS's main concept of colocating computation and data on the compute nodes: the bisection bandwidth across the compute nodes (e.g., 3D torus) is higher than the interconnect between the compute nodes and the storage nodes (e.g., Ethernet fat-tree).

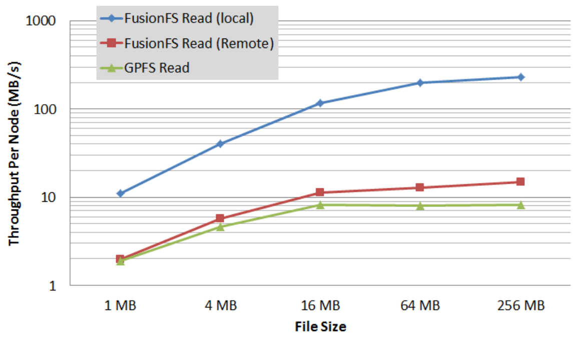


Figure 10. Read throughput of FusionFS and GPFS on Intrepid

In practice, the read throughput is somewhere between the two bounds, depending on the access pattern of the application and whether there is a data-aware scheduler to optimize the task placement. FusionFS exposes this much-needed data locality (via the metadata service) in order for parallel programming systems (e.g., Swift [33]) and job scheduling systems (e.g., Falkon [37]) to implement the data-aware scheduling. Note that Falkon has already implemented a data-aware scheduler for the "data diffusion" storage system [37], a precursor to the FusionFS project that lacked distributed metadata management, hierarchical directory-based namespace, and POSIX support. One potential improvement to FusionFS's read throughput lies in better algorithms for predicting future I/O patterns; we plan to explore this direction with incremental algorithms such as [38–40].

One might argue that FusionFS outperforms GPFS mainly because FusionFS is a distributed file system on compute nodes and the bandwidth is higher than the network between the compute nodes and the storage nodes. Again, we emphasize that FusionFS was designed specifically to take advantage of the fast interconnects across the compute nodes. Nevertheless, we note that FusionFS's unique I/O strategy also plays a critical role in reaching the high and scalable throughput, as discussed in Section IV-C. Thus one might more fairly compare FusionFS with other distributed file systems in the same hardware, architecture, and configuration. To this end, we deployed FusionFS and HDFS [11] on the Kodiak [41] cluster. We chose the Kodiak because Intrepid does not support Java (required by HDFS).

Kodiak is a 1024-node cluster at Los Alamos National Laboratory. Each Kodiak node has an AMD Opteron 252 CPU (2.6 GHz), 4 GB RAM, and two 7200 rpm 1 TB hard drives. In this experiment, each client of FusionFS and HDFS writes 1 GB data to the file system. Both file systems set replica to 1 in order to achieve the highest possible performance, and both turn off the FUSE interface.

Figure 11 shows that the aggregate throughput of FusionFS outperforms HDFS by about an order of magnitude. FusionFS shows excellent scalability, whereas HDFS starts to taper off at 256 nodes, mainly because of the weak write locality as data chunks (64 MB) need to be scattered out to multiple remote nodes.

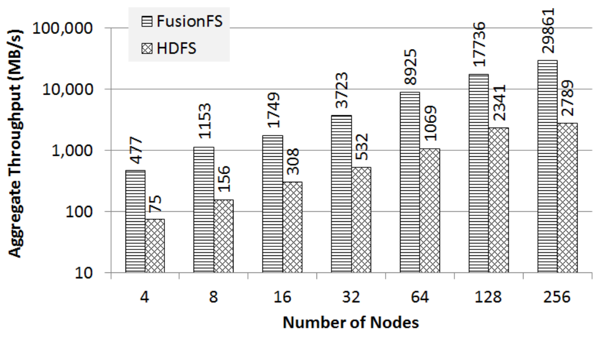
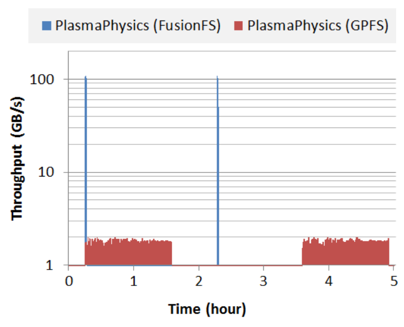
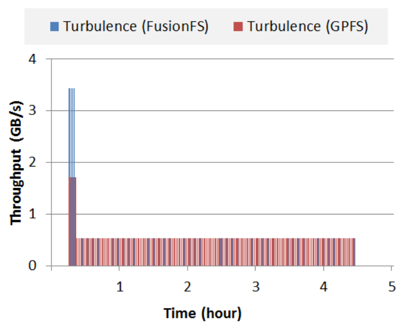


Figure 11. Throughput of FusionFS and HDFS on Kodiak

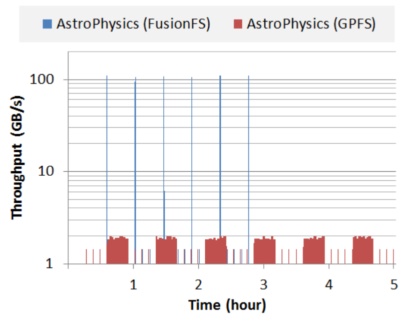
Clearly, FusionFS is not intended to compete with HDFS



(a) PlasmaPhysics



(b) Turbulence



(c) AstroPhysics

Figure 12. Top three write-intensive applications on Intrepid

but to target the scientific applications on HPC machines that HDFS was not originally designed for or even suitable for. Thus we had to restrict our design to fit the typical HPC machine specification: a massive number of homogeneous and less-powerful cores with limited per-core RAM. And, for a fair comparison, we had to deploy HDFS on the same hardware, which may or may not be an ideal or optimized testbed for HDFS.

C. Applications

We are interested in, quantitatively, how FusionFS helps reduce the I/O cost for real applications. To this end, we evaluated four scientific applications using FusionFS on Intrepid. The performance was compared mainly with Intrepid's default storage, the GPFS [14] parallel file system.

For the first three applications, we replayed the top three write-intensive applications on Intrepid [15] in December 2011 [5] on FusionFS: PlasmaPhysics, Turbulence, and AstroPhysics. While the PlasmaPhysics makes significant use of unique file(s) per node, the other two write to shared files. FusionFS is a file-level distributed file system, so PlasmaPhysics is a good example to benefit from FusionFS. However, FusionFS does not provide good N-to-1 write support for Turbulence and AstroPhysics. To make FusionFS's results comparable with those of GPFS for Turbulence and AstroPhysics, we modified both workloads to write to unique files as the exclusive chunks of the share file. Because of limited space, only the first five hours of these applications running on GPFS are considered here.

Figure 12 shows the real-time I/O throughput of these workloads at 1,024 nodes. On FusionFS, these workloads are completed in 2.38, 4.97, and 3.08 hours for PlasmaPhysics, Turbulence, and AstroPhysics, respectively. Recall that all these workloads are completed in 5 hours in GPFS.

We note that for both the PlasmaPhysics and AstroPhysics applications, the peak I/O rates for GPFS top at around 2 GB/s, whereas for FusionFS they reach over 100 GB/s. This increase in I/O performance accelerates the applications 2.1X times (PlasmaPhysics) and 1.6X times (AstroPhysics). The reason Turbulence does not benefit much from FusionFS is that this application does not have many consecutive I/O operations and GPFS is sufficient for such workload patterns: the heavy interleaving of I/O and computation does not push much I/O pressure to the storage system.

The fourth application, Basic Local Alignment Search Tool (BLAST), is a popular bioinformatics application to benchmark parallel and distributed systems. BLAST searches one or more nucleotide or protein sequences against a sequence database and calculates the similarities. It has been implemented with different parallelized frameworks, for example, ParallelBLAST [42]. In ParallelBLAST, the entire database (4 GB) is split into smaller chunks on different nodes. Each node then formats its chunk into an encoded slice and searches protein sequence against the slice. All the search results are merged together into the final matching result.

We compared ParallelBLAST performance on FusionFS and GPFS with our AME (Any-scale MTC Engine) framework [43]. We carried out a weak-scaling experiment of ParallelBLAST with 4 GB database on every 64 nodes and increased the database size proportionally to the number of nodes. The application has three stages (formatdb, blastp, and merge), which produce an overall data I/O of 541 GB over 16,192 files for every 64 nodes. In our experiment of 1024-node scale, the total I/O is about 9 TB applied to over 250,000 files.

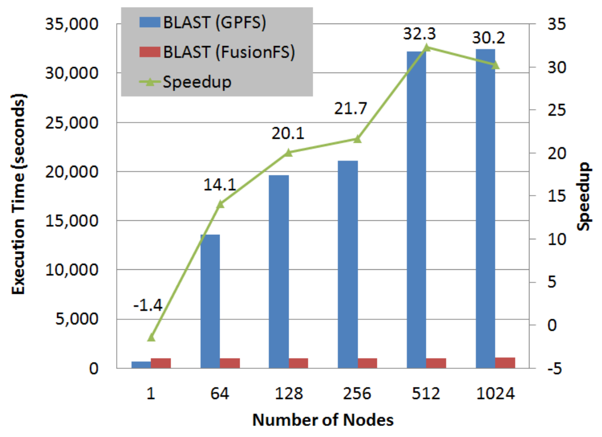


Figure 13. BLAST execution time on Intrepid

Figure 13 shows a huge (more than one order of magnitude) performance gap between FusionFS and GPFS at all scales except for the trivial 1-node case. FusionFS has up to 32X speedup (at 512 nodes), and an average of 23X improvement between 64 nodes and 1,024 nodes. At the 1-node scale, the GPFS kernel module is more effective in accessing an idle parallel file system. In FusionFS's case, the 1-node scale result involves the user-level FUSE module, which apparently causes BLAST to run 1.4X slower on FusionFS. However, beyond the corner case of 1 node, FusionFS significantly outperforms GPFS. In particular, the 1024-node BLAST requires 1,073 seconds to complete all three stages on FusionFS, whereas it needs 32,440 seconds to complete the same workload on GPFS.

VI. Related Work

Researchers have produced many shared and parallel file systems, such as the Network File System (NFS [44]), General Purpose File System (GPFS [14]), Parallel Virtual File System (PVFS [16]), Lustre[20], and Panasas[13]. These systems assume that storage nodes are significantly fewer than the compute nodes and that compute resources are agnostic of the data locality on the underlying storage system, thus resulting in an unbalanced architecture for data-intensive workloads.

Various distributed file systems have been developed, such as Google File System (GFS [29]), Hadoop File System (HDFS [11]), Ceph [45], and Sector [46]. However, many of these file systems are tightly coupled with execution frameworks (e.g., MapReduce [47]), with the result that scientific applications not using these frameworks must be modified to use these non-POSIX file systems. Moreover, those that do offer a POSIX interface are not designed for metadata-intensive operations at extreme scales. The majority of these systems do not expose the data locality information for general computational frameworks (e.g., batch schedulers, workflow systems) to harness the data locality through data-aware scheduling. In short, these distributed file systems are not designed specifically for HPC and scientific computing workloads and the scales anticipated by HPC in the coming years.

The idea of distributed metadata can be traced back to xFS [48], even though a central manager is needed in order to locate a particular file. Recently, FDS [49] was proposed as a blob store on data centers. It maintains a lightweight metadata server and offloads the metadata to available nodes in a distributed manner. In contrast, FusionFS metadata is completely distributed without any single-point-of-failure involved.

Colocation of compute and storage resources has attracted considerable research interest. For instance, Salus [50] proposes to colocate the storage to data nodes in data centers. Other examples include Rhea [51], which prevents removing the data used by the computation, and Nectar [52], which automatically manages data and computation in data centers. While these systems apply a general rule to deal with data I/O, FusionFS is optimized for write-intensive workloads that are particularly important for HPC systems.

VII. Conclusion and Future Work

This paper proposes a distributed storage layer on compute nodes to tackle the HPC I/O bottleneck of scientific applications. We identify the challenges this unprecedented architecture brings, and we build a distributed file system FusionFS to tackle them. In particular, FusionFS is crafted to support extremely intensive metadata operations and is optimized for file writes. Extreme-scale evaluation on up to 16K nodes demonstrates FusionFS's superiority over other popular storage systems for scientific applications.

We plan to explore the feasibility of integrating memory-centric middleware (e.g., Tachyon [53]) for cooperative caching [54]. We also plan to investigate better support for emerging workloads in HPC applications such as many-task computing [55].

Acknowledgments

This work was supported in part by the National Science Foundation under awards OCI-1054974 (CAREER). This material was also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computer Research, under contract number DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This material also is supported in part by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE) [56]. http://www.nmc-probe.org/

References

[1] I. Raicu, I. Foster, A. Szalay, and G. Turcu, "AstroPortal: A science gateway for large-scale astronomy data analysis," in *TeraGrid Conference*, June 2006.

[2] P. Freeman, D. Crawford, S. Kim, and J. Munoz, "Cyberinfrastructure for science and engineering: Promises and challenges," *Proceedings of the IEEE*, vol. 93, no. 3, 2005.

[3] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Using simulation to explore distributed key-value stores for extreme-scale system services," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[4] D. Zhao, D. Zhang, K. Wang, and I. Raicu, "Exploring reliability of exascale systems through simulations," in *Proceedings of the 21st ACM/SCS High Performance Computing Symposium (HPC)*, 2013.

[5] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies*, 2012.

[6] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the duality of data-intensive file system design: Reconciling HDFS and PVFS," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[7] D. Zhao and I. Raicu, "Distributed file systems for exascale computing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12), doctoral showcase*, 2012.

[8] DEEP-ER, "http://www.hpc.cineca.it/projects/deep-er," Accessed September 5, 2014.

[9] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *Proceedings of IEEE International Conference on Cluster Computing*, 2012.

[10] Mira, "https://www.alcf.anl.gov/user-guides/mira-cetus-vesta," Accessed September 5, 2014.

[11] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[12] B. Welch and G. Noer, "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions," in *Mass Storage Systems and Technologies, 2013 IEEE 29th Symposium on*, 2013.

[13] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *Proceedings of ACM/IEEE Conference on Supercomputing*, 2004.

[14] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[15] Intrepid, "https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor," Accessed September 5, 2014.

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

[17] D. Zhao, J. Yin, K. Qiao, and I. Raicu, "Virtual chunks: On supporting random accesses to scientific data in compressible storage systems," in *Proceedings of IEEE International Conference on Big Data*, 2014.

[18] D. Zhao, J. Yin, and I. Raicu, "Improving the i/o throughput for data-intensive scientific applications with efficient compression mechanisms," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13), poster session*, 2013.

[19] FUSE, "http://fuse.sourceforge.net," Accessed September 5, 2014.

[20] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," in *Proceedings of the linux symposium*, 2003.

[21] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proceedings of ACM Symposium on Applied Computing*, 2010.

[22] D. Zhao and I. Raicu, "HyCache: A user-level caching middleware for distributed file systems," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013.

[23] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, 2013.

[24] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu, "Exploring distributed hash tables in highend computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 3, Dec. 2011.

[25] D. Zhao, C. Shou, T. Malik, and I. Raicu, "Distributed data provenance for large-scale data-intensive computing," in *Cluster Computing, IEEE International Conference on*, 2013.

[26] C. Shou, D. Zhao, T. Malik, and I. Raicu, "Towards a provenance-aware distributed filesystem," in *5th Workshop on the Theory and Practice of Provenance (TaPP)*, 2013.

[27] D. Zhao, K. Burlingame, C. Debains, P. Alvarez-Tabio, and I. Raicu, "Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms," in *Cluster Computing, IEEE International Conference on*, 2013.

[28] Protocol Buffers, "http://code.google.com/p/protobuf/," Accessed September 5, 2014.

[29] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[30] epoll, "http://man7.org/linux/man-pages/man7/epoll.7.html," Accessed September 5, 2014.

[31] Y. Gu and R. L. Grossman, "Supporting configurable congestion control in data transport services," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[32] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, S. Kenny, K. Iskra, P. Beckman, and I. Foster, "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers," *Journal of Physics: Conference Series*, vol. 180, no. 1, 2009.

[33] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Proceedings of the 2007 IEEE Congress on Services*, 2007.

[34] I. Raicu, I. T. Foster, Y. Zhao, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain, "The quest for scalable support of data-intensive workloads in distributed systems," in *Proceedings of ACM International Symposium on High Performance Distributed Computing*, 2009.

[35] Top500, "http://www.top500.org/list/2014/06/," Accessed September 5, 2014.

[36] ZeptoOS, "http://www.mcs.anl.gov/zeptoos," Accessed September 5, 2014.

[37] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a fast and light-weight task execution framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[38] D. Zhao and L. Yang, "Incremental isometric embedding of high-dimensional data using connected neighborhood graphs," *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, vol. 31, no. 1, Jan. 2009.

[39] R. Lohfert, J. Lu, and D. Zhao, "Solving sql constraints by incremental translation to sat," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, 2008.

[40] D. Zhao and L. Yang, "Incremental construction of neighborhood graphs for nonlinear dimensionality reduction," in *Proceedings of International Conference on Pattern Recognition*, 2006.

[41] Kodiak, "https://www.nmc-probe.org/wiki/machines:kodiak," Accessed September 5, 2014.

[42] D. R. Mathog, "Parallel BLAST on split databases," *Bioinformatics*, vol. 19(4), pp. 1865 – 1866, 2003.

[43] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. T. Foster, "Design and analysis of data management in scalable parallel scripting," in *Proceedings of ACM/IEEE conference on Supercomputing*, 2012.

[44] M. Eisler, R. Labiaga, and H. Stern, "Managing NFS and NIS, 2nd ed." *O'Reilly & Associates, Inc.*, 2001.

[45] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[46] Y. Gu, R. L. Grossman, A. Szalay, and A. Thakar, "Distributing the Sloan Digital Sky Survey using UDT and Sector," in *Proceedings of IEEE International Conference on e-Science and Grid Computing*, 2006.

[47] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of USENIX Symposium on Opearting Systems Design & Implementation*, 2004.

[48] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless network file systems," in *Proceedings of ACM symposium on Operating systems principles*, 1995.

[49] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat datacenter storage," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[50] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin, "Robustness in the salus scalable block store," in *Proceedings of USENIX conference on Networked Systems Design and Implementation*, 2013.

[51] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. Rowstron, "Rhea: automatic filtering for unstructured cloud storage," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, 2013.

[52] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: automatic management of data and computation in datacenters," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[53] Tachyon, "http://tachyon-project.org/," 2014.

[54] D. Zhao, K. Qiao, and I. Raicu, "Hycache+: Towards scalable high-performance caching middleware for parallel file systems," in *Proceedings of IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[55] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain, *Towards Data Intensive Many-Task Computing*, T. Kosar, Ed. IGI Global, 2012.

[56] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, "Probe: A thousand-node experimental cluster for computer systems research," *USENIX ;login:*, vol. 38, no. 3, June 2013.