BIG DATA SYSTEM INFRASTRUCTURE AT EXTREME SCALES

BY

DONGFANG ZHAO

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _______________________
Advisor

Chicago, Illinois
July 2015

ACKNOWLEDGMENT

This dissertation could not have been completed without the consistent encouragement and support from my advisor Dr. Ioan Raicu and my family. I am grateful to Dr. Robert Ross and Dr. Jian Yin for advising my research work at Argonne National Laboratory and Pacific Northwest National Laboratory, respectively. There are many more people who help me succeed through my PhD study, particularly my dissertation committee: Dr. Xian-He Sun, Dr. Zhiling Lan, and Dr. Erdal Oruklu.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Rapid advances in digital sensors, networks, storage, and computation along with their availability at low cost is leading to the creation of huge collections of data – dubbed as Big Data. This data has the potential for enabling new insights that can change the way business, science, and governments deliver services to their consumers and can impact society as a whole. This has led to the emergence of the Big Data Computing paradigm focusing on sensing, collection, storage, management and analysis of data from variety of sources to enable new value and insights. To realize the full potential of Big Data Computing, we need to address several challenges and develop suitable conceptual and technological solutions for dealing them. Today's and tomorrow's extreme-scale computing systems, such as the world's fastest supercomputers, are generating orders of magnitude more data by a variety of scientific computing applications from all disciplines. This dissertation addresses several big data challenges at extreme scales. First, we quantitatively studied through simulations the predicted performance of existing systems at future scales (for example, exascale 10^{18} flops). Simulation results suggested that current systems would likely fail to deliver the needed performance at exascale. Then, we proposed a new system architecture and implemented a prototype that was evaluated on tens of thousands nodes on par with the scale of today's largest supercomputers. Micro benchmarks and real-world applications demonstrated the effectiveness of the proposed architecture: the prototype achieved up to two orders of magnitude higher data movement rate than existing approaches. Moreover, the system prototype was incorporated with features that were not well supported in conventional systems, such as distributed metadata management, distributed caching, lightweight provenance, transparent compression, acceleration through GPU encoding, and parallel serialization. Towards exploring the proposed architecture at millions of node scales, simulations were conducted and evaluated with a variety of workloads, showing near linear scalability and orders of

magnitude better performance than today's state-of-the-art storage systems.

CHAPTER 1
INTRODUCTION

We are in the era of big data. More and more of our daily activities are being digitalized whose data are conveniently recorded by and accessible from many sorts of devices ranging from mobile phones, to tablets, to laptops. For instances, we do not have to call a taxi, we do not have to print a flight boarding pass, and we do have to pay at the parking meter on the street; all of these can be done by a couple of taps on our smart phones. Nevertheless, most of the data curation and data analytics happen not on the client devices but on a remote *facility*. This *facility* is usually a distributed computing system such as a cluster of tens of blade servers or a cloud comprised of hundreds of virtual machines. How to make the facility scalable to the unprecedented volume of data is categorically a challenging problem.

Another major reason of such a data explosive is the technological advance of computing chips. The computing capability of CPUs still follows Moore's law [124]: the per-chip flops has been doubling every 12 – 18 months (firstly by boosting the frequency, then by increasing the number of cores). Thanks to the emergence of powerful computing chips, many applications that used to be throttled by computing capability are able to process and output data exponentially faster than before. The performance bottleneck, thus, now lies in the data side as opposed to the computing side.

The emergence of big data inevitably impacts many conventional computing paradigms such as high performance computing (HPC) and supercomputing that involve the most powerful machines worldwide (i.e., TOP500 supercomputers [188]). These systems are today's leading computing power (i.e., petascale 10^15 flops) aggregated from tens of thousands of CPUs and hundreds of thousands of cores. Nevertheless, the architecture of HPC and supercomputing systems has barely changed for the last decade and shows limitations for modern data-intensive applications. In particular, HPC and supercomputing systems separate the compute and storage resources into two cliques (compute nodes and storage nodes), both of which are interconnected by a shared network. This architecture is mainly a result of the nature of many legacy large-scale applications that are compute-intensive, where it was often assumed that the storage I/O capabilities were lightly utilized only for initial data input, occasional checkpoints, and final output. Therefore, as the bottleneck used to be the computation, significantly more resources were invested in the computational capabilities of these systems. This is, unfortunately, not true in the era of big data.

Modern applications at extreme scales are becoming data-centric [76]; workloads are now data-intensive rather than compute-intensive, requiring a greater degree of support from the storage subsystem [58]. Recent studies (for example, [107, 29]) address the I/O bottleneck of the conventional architecture of HPC and supercomputing systems. Liu et. al. [107] proposed a middleware layer in between the storage and compute nodes (i.e., I/O nodes) that deal with the I/O bursts. Carns et. al. [29] proposed several techniques to optimize the I/O performance of accessing small file in the conventional parallel file systems. All aforementioned studies focused on addressing the data bottleneck of conventional HPC and supercomputing system architectures.

This work is orthogonal to existing studies on big data system infrastructure at extreme scales: we propose a new architecture that collocates storage with compute resources. In particular, we envision a distributed storage system on compute nodes for applications to manipulate their intermediate results and checkpoints; the data only need to be transferred over the network to the remote storage for archival purposes. While co-location of storage and computation has been widely adopted in cloud computing and data centers (for example, Hadoop clusters [14]), such architecture has not been well studied in HPC and supercomputing systems despite it attracting a lot of research interest (for example, the DEEP-ER [45] project). This work is one of the pioneer works that demonstrates how to architect and engineer such a system, and reports how much, quantitatively, it could improve the performance of real-world scientific applications at today's petascale and future exascale computing systems.

This dissertation will discuss the following topics. In Chapter 2, we conduct extensive simulation work [233] to illustrate that the conventional architecture of HPC and supercomputing systems would not be viable for future scales such as exascale computing. We then propose a new system architecture [225] and implement a prototype (FusionFS [234]) in Chapter 3. On top of FusionFS, we design several unconventional features on distributed caching [226, 222, 223] (Chapter 4), lightweight provenance [177, 227] (Chapter 5), transparent compression [232, 231] (Chapter 6), GPU encoding [220] (Chapter 7), and parallel serialization (Chapter 8). In Chapter 9, we simulate the performance of FusionFS at exascale [221], i.e. on millions of nodes. Chapter 10 reviews related work in big data applications and data-intensive distributed systems at extreme scales. We finally conclude this dissertation in Chapter 11.

CHAPTER 2
LIMITATIONS OF THE CONVENTIONAL ARCHITECTURE

Exascale computers are predicted to emerge by the end of this decade with millions of nodes and billions of concurrent cores/threads. One of the most critical challenges for exascale computing is how to effectively and efficiently maintain the system reliability. Checkpointing is the state-of-the-art technique for high-end computing system reliability that has proved to work well for current petascale scales.

This chapter investigates the suitability of checkpointing mechanism for exascale computers, across both parallel filesystems and distributed filesystems. We built a model to emulate exascale systems, and developed a simulator, RXSim [233], to study its reliability and efficiency. Experiments show that the overall system efficiency and availability would go towards zero as system scales approach exascale with checkpointing mechanism on parallel filesystems. However, the simulations suggest that a distributed filesystem with local persistent storage would offer excellent scalability and aggregate bandwidth, enabling efficient checkpointing at exascale.

2.1 Conventional HPC Architecture

State-of-the-art storage subsystems for high-performance computing (HPC) are mainly comprised of the parallel filesystems (for example, GPFS [172]) deployed on remote storage servers. That is, the compute and storage resources are segregated, and interconnected through a shared commodity network. A typical HPC architecture is illustrated in Figure 2.1, where the network-attached storage (NAS) serves the I/O requests from the compute resource.

There are two main reasons why HPC systems are designed like this. First,

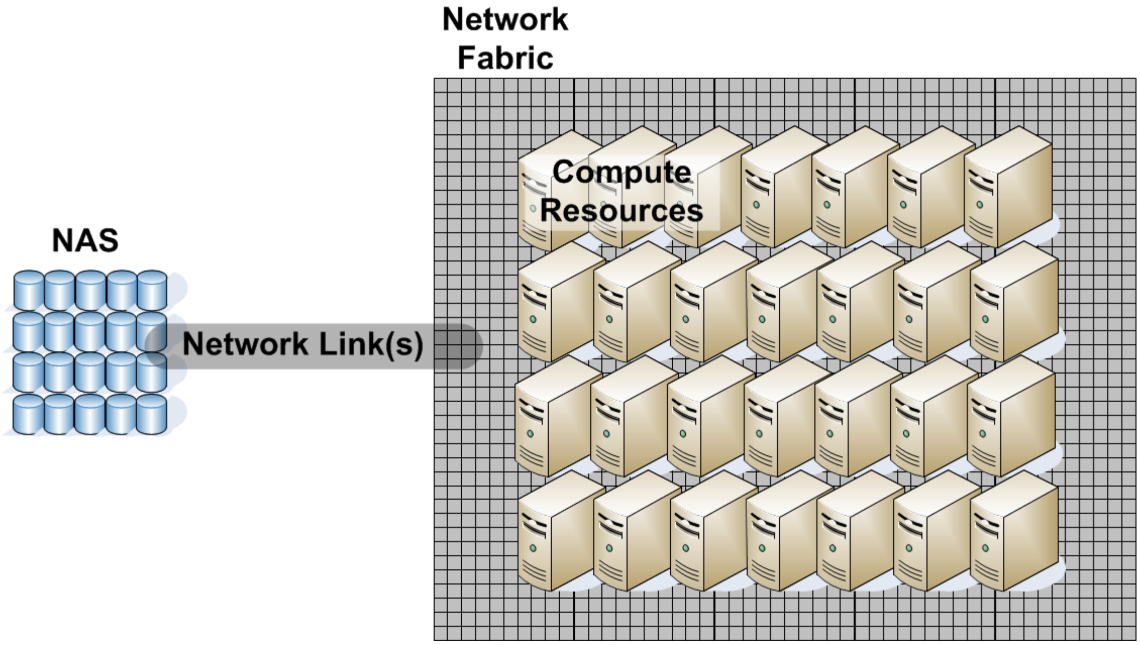


Figure 2.1. Conventional HPC architecture

many legacy scientific applications are compute-intensive, and barely touch the persistent storage except for initial input, occasional checkpointing, and final output. Therefore the shared network between compute and storage nodes does not become a performance bottleneck or single point of failure. Second, a parallel filesystem proves to be highly effective for the concurrent I/O workload commonly seen in scientific computing. In essence, a parallel filesystem splits a (big) file into smaller subsets so that multiple clients can access the file in parallel. Popular parallel filesystems include Lustre [173], GPFS [172], and PVFS [30].

While modern applications are becoming more data-intensive, researchers spend significant effort on improving the I/O throughput of the aforementioned architecture. Of note, recent studies [107, 184, 29] addressed the I/O bottleneck in the conventional architecture of HPC systems. Nevertheless, the theoretical bottleneck –the shared network between the compute and storage– will continue to exist.

2.2 Node-Local Storage for HPC

One straightforward means to address the bottleneck of conventional storage architecture is to eliminate the network. That is, the storage system is accessible right on the local disk. In particular, we envision a distributed filesystem deployed on compute nodes, as shown in Figure 2.2.

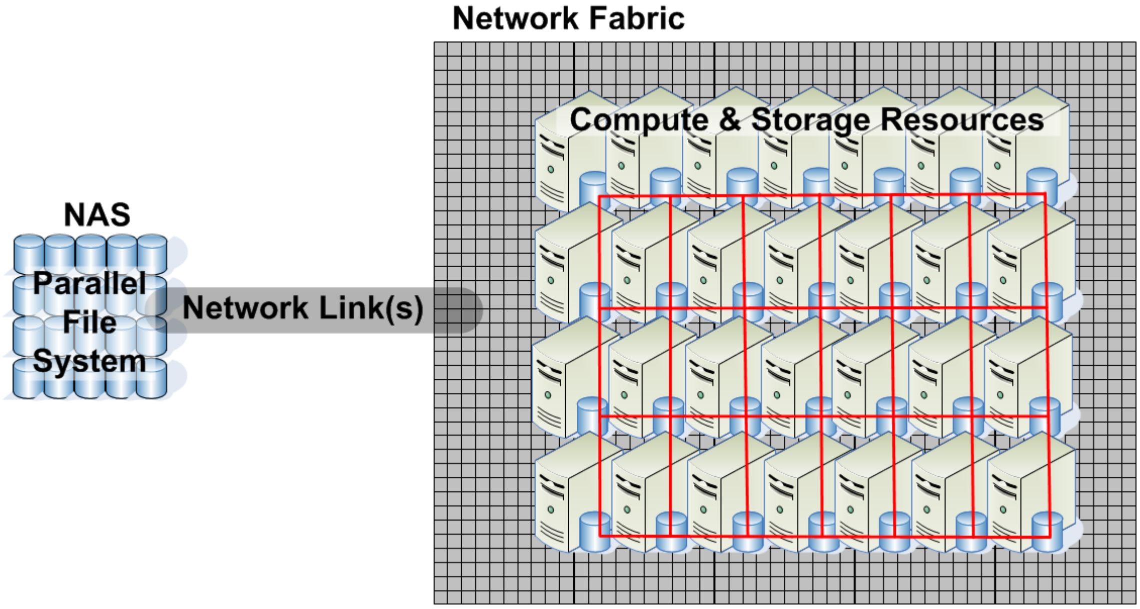


Figure 2.2. Co-location of compute and storage nodes

In the proposed architecture, the compute nodes are mingled with on-board hard disks, which, at the first glance, seems a retrofit of data centers. But there are some key differences between HPC and cloud computing. We list three of them in the following:

First, the interconnect within compute clusters is significant faster than data centers. It is not uncommon to see 3D-torus network in HPC systems, whereas data centers typically have Ethernet as their network infrastructure.

Second, the software stacks of HPC and data centers are different. Data centers take good advantage of virtual machines for better system utilization, support popular high-level programming language such as Java and Python. In contrast, HPC just cannot sacrifice performance to improve utilization, since the top mission of HPC systems is to accelerate large-scale simulations and experiments. More "modern" programming language such as Java and Python, are not well supported because most scientific applications are written in C and Fortran.

Last, and probably the most important reason is that, HPC and data centers target at different users and workloads. As a case in point, the Hadoop filesystem (HDFS [178]) in data centers deals with files of default 64 MB chunks (preferable 128 MB). This is because in data centers files are typically large in size. HPC applications, however, have many small- and medium-sized files, as Welch and Noer [199] reported that 25% – 90% of all the 600 million files from 65 Panasas [134] installations are 64 KB or smaller.

With all the above discrepancies among others, existing storage solutions in data centers are not optimized for HPC machines. Therefore a distributed filesystem crafted for HPC is in need. Nevertheless, before building a real distributed filesystems on HPC compute nodes it would be desirable to have simulations to justify our expectation and as co-design of system implementation.

2.3 Modeling HPC Storage Architecture

This section describes how we model the two major storage designs for large scale HPC systems: remote parallel filesystems and node-local distributed filesystems. In particular, we are interested in their checkpointing performance, one of the most I/O-intensive applications in HPC. Before that, we introduce some metrics and terminology.

Application Efficiency is defined as the ratio of application up time over the total running time:

$$E = \frac{up\_time}{running\_time} \times 100\%,$$

where *running_time* is the summation of up time, checkpointing time, lost time and rebooting time. **Up time** is when the job is correctly running on the computer. **Checkpointing time** is when the system stores the correct states on persistent storage periodically. **Lost time** measures the time when a failure occurred, the work since the last checkpointing would be lost and needs to be recalculated. **Rebooting time** is simply the time for the system to reboot the node.

Optimal Checkpointing Interval is the optimal checkpointing interval as modeled in [41]:

$$OPT = \sqrt{2\delta(M+R)} - \delta,$$

where δ is the checkpointing time, M is the system mean-time-to-failure (MTTF) and R is the rebooting time of a job.

Memory Per Node is modeled as the following based on the specifications of IBM Blue Gene/P. When the system has fewer than 64K nodes, each node has 2 GB memory. For larger systems, the per-node memory is calculated (in GB) as

$$2 \cdot \frac{\#nodes}{64K}$$

We have two different models of **Storage Bandwidth** for parallel filesystems (PFS) and distributed filesystems (DFS), respectively, since they have completely different architectures. We assume PFS is the state-of-the-art parallel filesystem used in production today, e.g. GPFS [172], whose bandwidth (in GB/sec) is modeled as

$$BW_{PFS} = \frac{\#nodes}{1000}$$

As for DFS, it is a hypothetical new storage architecture for exascale. There are no real implementations of a DFS that can scale to exascale, but this study should be a good motivator towards investing resources to the realization of DFS at exascale. The bandwidth of DFS in our simulation has the following bandwidth

$$BW_{DFS} = \#nodes \cdot (\log \#nodes)^2$$

These equations are based on our empirical observations on the IBM Blue Gene/P supercomputer.

For rebooting time, DFS has a constant time of 85 seconds because each node is independent to other nodes. For PFS, the rebooting time (in seconds) is calculated as the following:

$$\lceil 0.0254 \cdot \#nodes + 55.296 \rceil$$

which is also based on the empirical data of the IBM Blue Gene/P supercomputer. The above formulae indicate that DFS has a linear scalability of checkpointing bandwidth, whereas PFS only scales sub-linearly. The sub-linearity of PFS checkpoint bandwidth would prevent it from working effectively for exascale systems.

2.4 The RXSim Simulator

For any job running on an HPC system, RXSim has three states: running, repairing, restarting, as shown in Figure 2.3. This transmission works as follows: 1) when a job is running, repairing or restarting, if a failure occurs then the job will be hanged and enters repairing state; 2) after repaired, the job will restart, i.e. reboot nodes occupied by this job; 3) after the job completes, it restarts its nodes; 4) after restarting, if job is just repaired from a failure then the job continues its work; otherwise the job has completed its work.

RXSim is implemented in Java with roughly 2K lines of code, and will be released as an open source project. Some key modules include job management, node management, and time stamping. We will discuss each of them respectively in the following subsections.

2.4.1 Job Management. The job management module is used to keep tracks of any job-related information during the run time. Every job in the workload is an instance of the *Job* class. A job has common attributes like *jobID*, *walltime*, *size*,

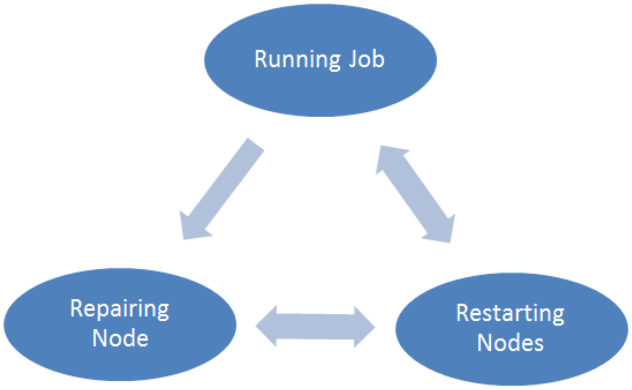


Figure 2.3. State transmission of RXSim.

endTime, and some state variables such as *state_up* and *state_repair*.

We do not keep the entire workload globally. Rather, each time the generator generates a new job, it is inserted into the running queue. Once a job is completed, the allocated nodes are restarted and released.

2.4.2 Node Management. Because the workload is randomly assigned to multiple work nodes and there may be many jobs running on the system at the same time, nodes need to have the information about which jobs are running on them. This is implemented by adding a *jobID* attribute to the *Node* class. Nodes management is analogous to traditional memory management.

An array is fulfilled with instances of *Node* class in order to keep all information such as node ID and working state. A free list is to keep and track all idle parts of the system, so that each time a job requests some computing resources (nodes) RXSim will first check if there are enough idle nodes left. If so, RXSim retrieves the first idle part of the system and keeps doing so till the job gets enough nodes. After a job is completed, the nodes occupied by this job will not be released immediately. These nodes would be occupied by the completed job until they are successfully rebooted.

2.4.3 Time Stamping.

TimeStamp class is the event class where each *timeStamp* instance is an event with some information related to time stamping. For example, when the simulator encounters a failure at some point it creates a *timeStamp* instance including the incident's time, type, node ID, and job ID.

There are four types of TimeStamp:

- Job ends successfully: with time and job ID.
- Job recovers: with time and job ID.
- Node reboots successfully: with time and job ID.
- Node failure: with time and node ID.

The TimeStamp queue is implemented as a TreeSet. The benefit of TreeSet is that it will automatically sort the data, so it is easy to retrieve the latest event from this queue. An obvious drawback of TreeSet is that its elements are hard to be modified. Unfortunately modification is a frequent operation since the simulator needs to update events in a regular basis. To fix the problem, we maintain another list called *uselessEventList*, which keeps tracks of all idle TimeStamp. The simulator would simply skip such an idle TimeStamp and try to retrieve the next available one.

2.5 Simulation Results

Experiments can be categorized into three major types. We first compare RXSim results to existing traces with the same parameters and workload to verify RXSim. Then variant workloads are dispatched to RXSim to study the effectiveness and efficiency of checkpointing at different scales. Lastly, we apply RXSim on a 8-month log of an IBM Blue Gene/P supercomputer and emulate the checkpointing at exascale. Metrics *Uptime*, *Check*, *Boot* and *Lost* refer to the definitions of **up time**, **checkpointing time**, **rebooting time** and **lost time**, respectively.

2.5.1 Experiment Setup. The single-node MTTF is set to 1000 years, optimistically, as claimed by IBM. We assume it takes 0 second (again, very optimistically) to repair a single node. Simulation time is set to 5 years, where each time step is 1 second.

2.5.2 Validation. Raicu et. al. [157] show how the applications look like for 3 cases: No Checkpointing, PFS with checkpointing, and DFS with checkpointing, as shown in Figure 2.4.

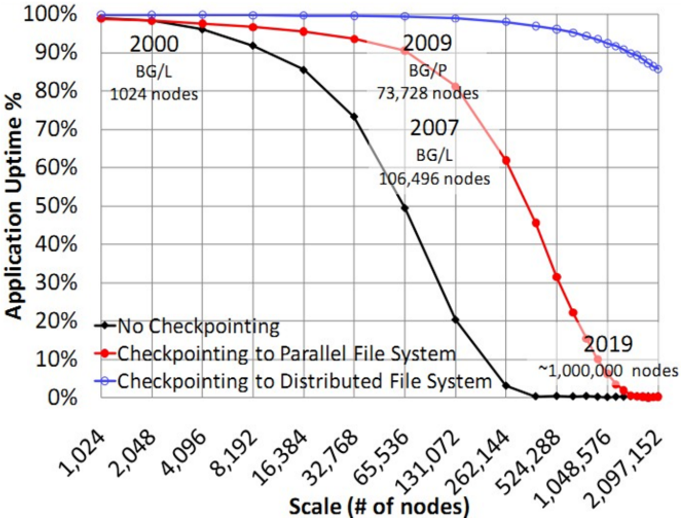


Figure 2.4. Comparison between checkpointings on different architectures [157]

The result of RXSim with the same workload of Figure 2.4 is shown in Figure 2.5. RXSim result is highly close to the published results: two lines have negligible difference, which is only due to the random variables used in the simulator.

Figure 2.6 shows the system reliability with checkpointing disabled. As we can see, the system is basically not functioning beyond 400K nodes.

Figure 2.7 shows the system reliability when enabling checkpointing on a PFS.

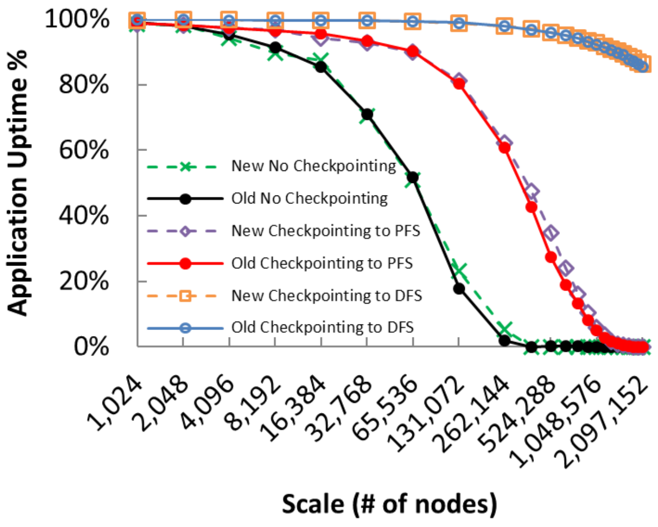


Figure 2.5. Comparison between RXSim and real-world traces

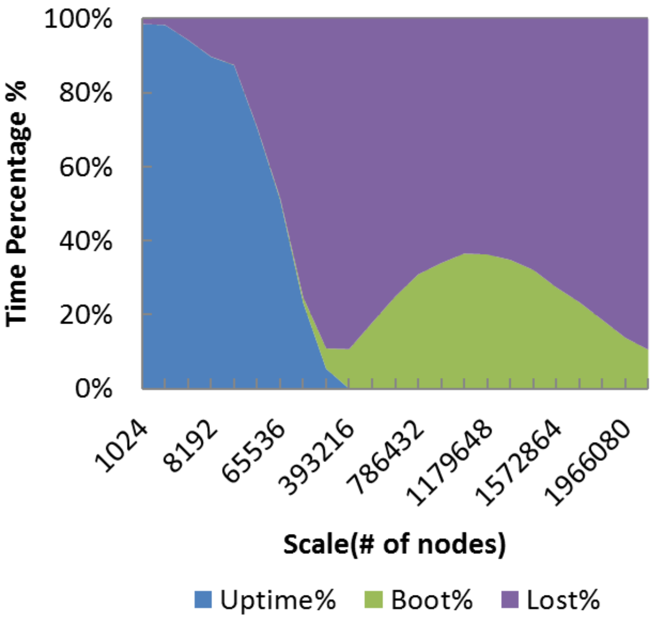


Figure 2.6. A no-checkpointing system stops functioning beyond 400K nodes

We observe that the system up time is significantly longer than Figure 2.6. This is expected, since checkpointing proves to be an effective mechanism to improve system reliability. However, the efficiency is extremely low (< 10%) at exascale (1 million nodes), meaning that PFS is not a good choice for checkpointing.

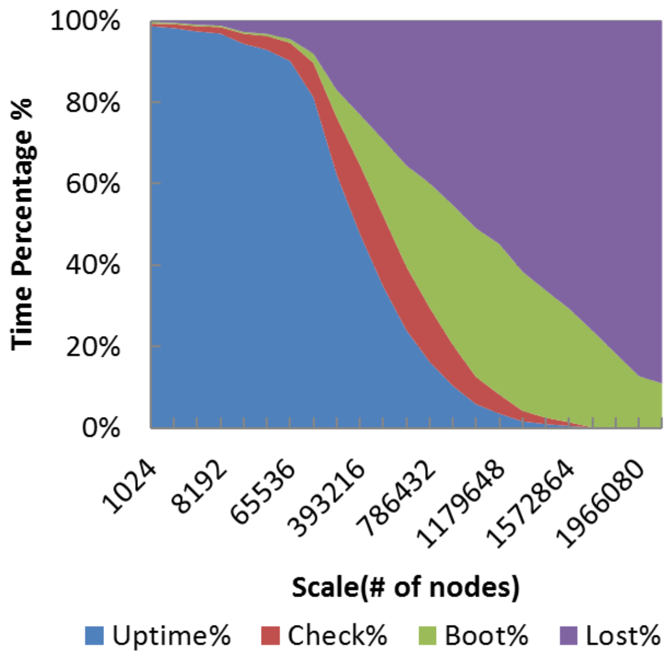


Figure 2.7. System reliability when enabling checkpointing on a PFS

In Figure 2.8, we show the trends of system MTTF, the overhead of doing a checkpoint, and the checkpointing circle (summation of checkpoint time and optimal checkpointing interval) in PFS. When system MTTF becomes shorter than the checkpointing circle time (which is the case for 1 million nodes), it basically means the system does not have enough time to complete one round of checkpointing. In other words, the system cannot recover from failures: checkpointing helps nothing but adding more overhead.

For DFS with checkpointing, we see an excellent application efficiency and scalability, as shown in Figure 2.9. The uptime portion is retained as high as 90% at exascale.

As shown in Figure 2.10, DFS is perfectly fine to allocate enough time slice for checkpointing at exascale. This can be best explained by the fact that DFS has less checkpointing overhead when writing to the local storage as opposed to NAS

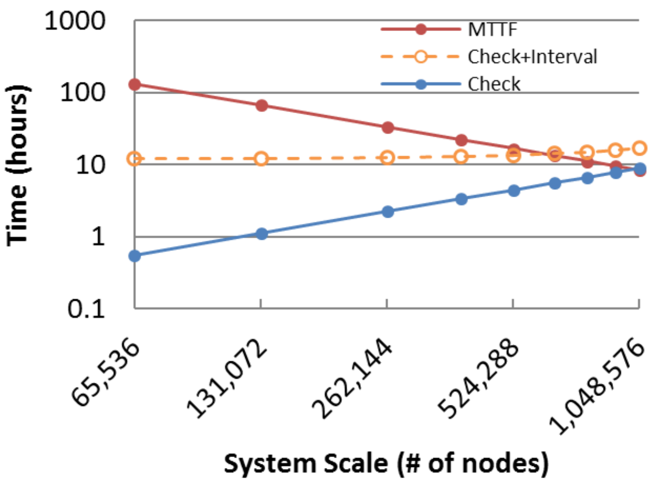


Figure 2.8. Trends of System MTTF, the overhead of doing a checkpoint, and the checkpointing circle

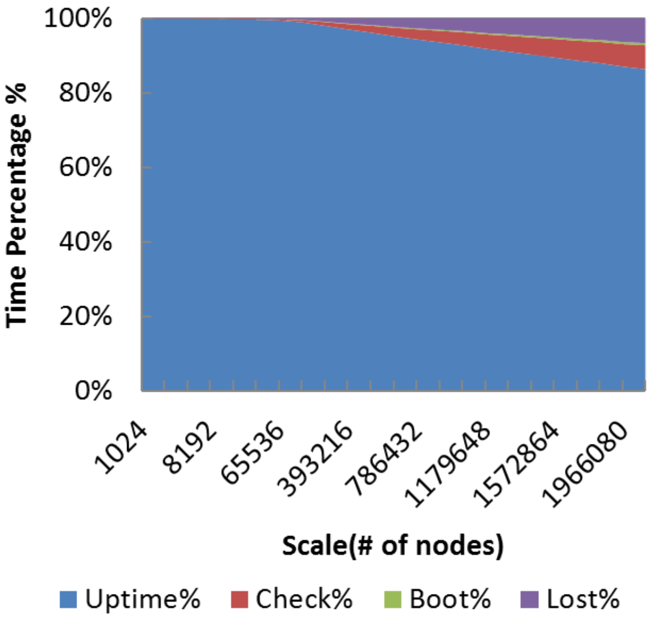


Figure 2.9. System reliability when enabling checkpointing on a DFS

(networked attached storage).

2.5.3 Synthetic Workloads.

We carried out two more workloads with different

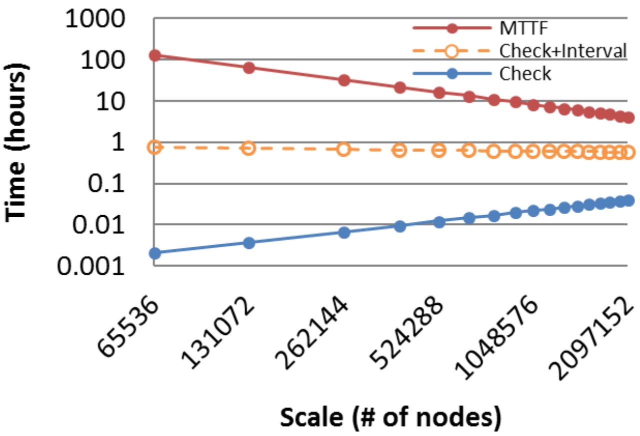


Figure 2.10. System reliability when enabling checkpointing on a DFS

job size and wall time on RXSim in this subsection.

2.5.3.1 1/10 Job Size. In this workload, each job size is 1/10 of the full system scale and the wall time is set to 7 days. The results are shown in Figure 2.11. The efficiency is generally better than the full system scale (see Figure 2.5). In particular, PFS with checkpointing on 1 million nodes is improved from 5% to 70%. This result demonstrates that the major bottleneck of PFS is the shared storage between jobs. The more concurrent jobs trying to access the shared storage, the less efficient PFS becomes.

We will show more details of time breakdown for each case. For no checkpointing, not surprisingly, the major two portion of costs are up time and lost time, as shown in Figure 2.12.

The cost breakdown for PFS with checkpointing is shown in Figure 2.13. Now the up time and checkpointing overhead are the top two portions, as expected.

We further examine how PFS with checkpointing would behave for system recovery. As shown in Figure 2.14, there is still a significant gap between MTTF and checkpointing time, which suggests that PFS with checkpointing might work well for

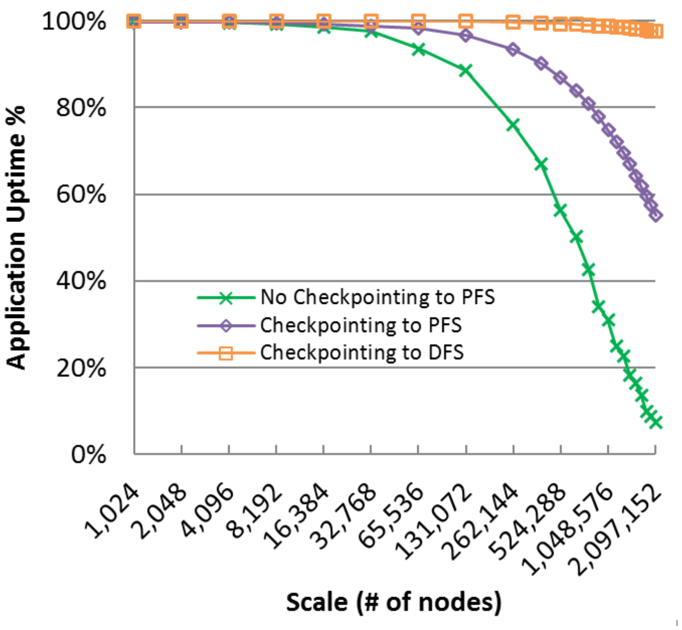


Figure 2.11. Comparison between different checkpointings with 1/10 job size

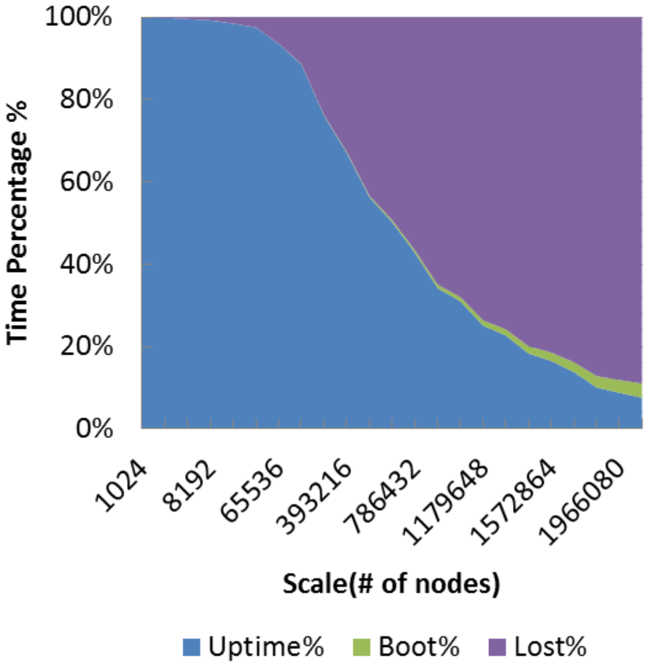


Figure 2.12. Cost breakdown of a no-checkpointing filesystem with 1/10 job size

1/10 job size. In particular on 1 millions nodes, the MTTF is about 10 hours and the checkpointing takes only 1 hour.

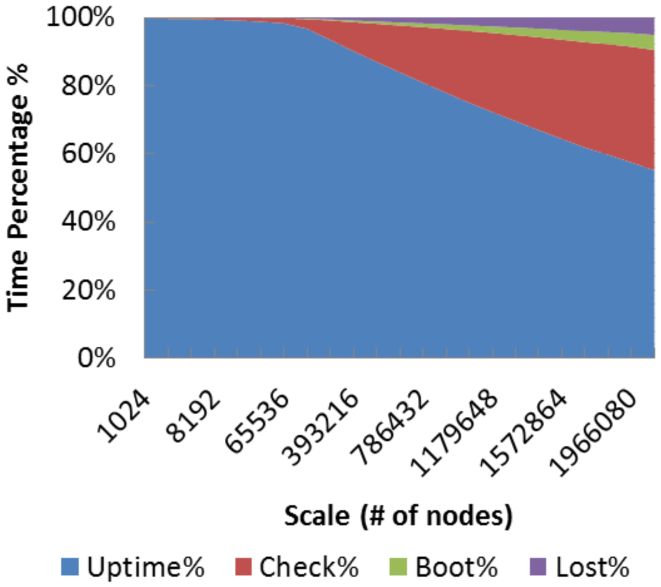


Figure 2.13. Cost breakdown of a PFS with checkpointing for 1/10 job size

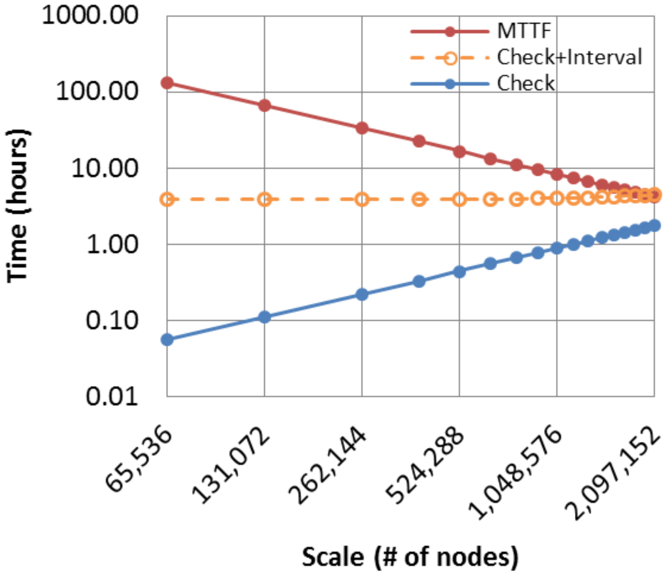


Figure 2.14. Trends of MTTF and checkpointing time for PFS

At last we show how DFS deals with 1/10 job size by plotting the cost breakdown. The result is shown in Figure 2.15. The up time is dominant and keeps taking over 95% percentage even for 2 million nodes.

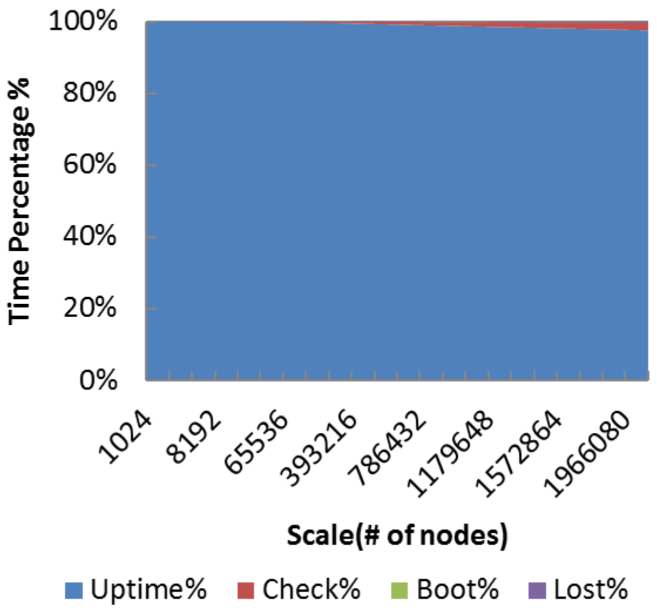


Figure 2.15. Cost breakdown of the DFS with checkpointing for 1/10 job size

2.5.3.2 One-Day Wall Time. This workload keeps the job size as the full system scale but shortens the wall time from 7 days to 1 day. We compare these relatively short jobs in 3 different filesystems as shown in Figure 2.16. Again, DFS outperforms other two and keeps high efficiency of 90% on 1 million nodes. Meanwhile, PFS is degenerated to be worse than no-checkpointing.

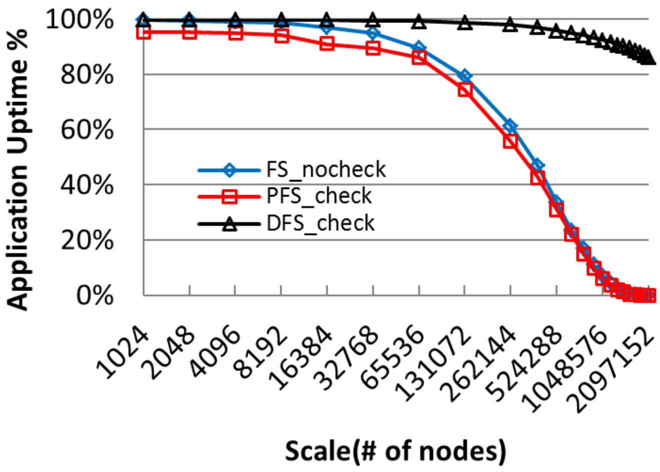


Figure 2.16. Efficiency of different checkpointing scenarios for short jobs

We show the cost breakdown of PFS and no-checkpointing in Figure 2.17 and Figure 2.18, respectively, in order to investigate why PFS has such poor performance. The cost distributions of both cases are about the same, except that PFS has some additional time spent on checkpointing that only takes a small portion (< 10%). The reason is that the wall time of each job is significantly shorter, which implies less lost-time during a failure. The checkpointing interval and checkpointing overhead are sensitive to wall time, therefore shortening jobs dramatically reduces the application efficiency of PFS with checkpointing. The implication, in order, is that for short jobs no-checkpointing might do as equally well as PFS with checkpointing enabled.

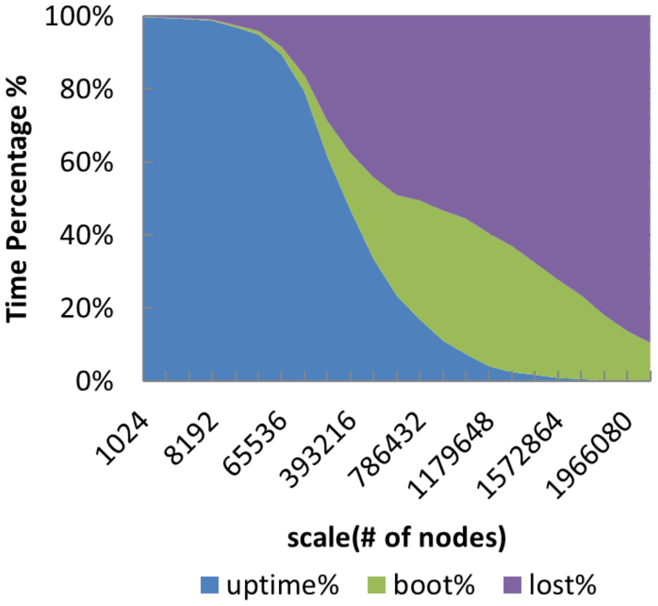


Figure 2.17. Cost breakdown of no-checkpointing filesystem for 1-day jobs

2.5.4 Real Logs of IBM Blue Gene/P. We carried out experiments on real workloads (8-month log) from an IBM Blue Gene/P supercomputer (Intrepid [79]) at Argonne National Laboratory. Intrepid has a peak of 557 TFlops, has 40 racks, and comprises 40,960 quad-core nodes (163,840 cores in total), 640 associated I/O nodes, 128 storage servers (NAS), and a 3-dimensional high bandwidth torus network interconnecting compute nodes. It debuted as No.3 in the top 500 supercomputer list

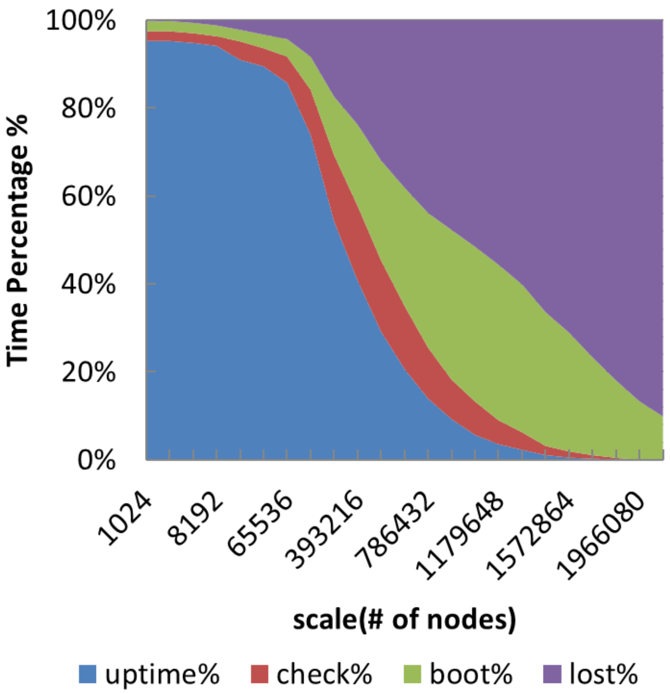


Figure 2.18. Cost breakdown of PFS with checkpointing for 1-day jobs

released in June 2008.

The log in the experiment contains 8 months of accounting records on Intrepid. The log data is in a format called swf (standard workload format). We scaled the job size on the log in proportion to the scale of RXSim from 1024 to 2 million nodes. Note that the data beyond 40K nodes are predicted by RXSim, since the Blue Gene/P only has 40K nodes (160K cores).

Figure 2.19 shows that a no-checkpointing filesystem outperforms PFS with checkpointing, which is counter intuitive at the first glance. The reason is that the Blue Gene/P jobs have an average wall time of 5k seconds, which is less than 2 hours and significantly shorter than 1 day in Figure 2.16. So this result in fact justifies our previous conclusion that short jobs would badly hurt the application efficiency by enabling checkpointing.

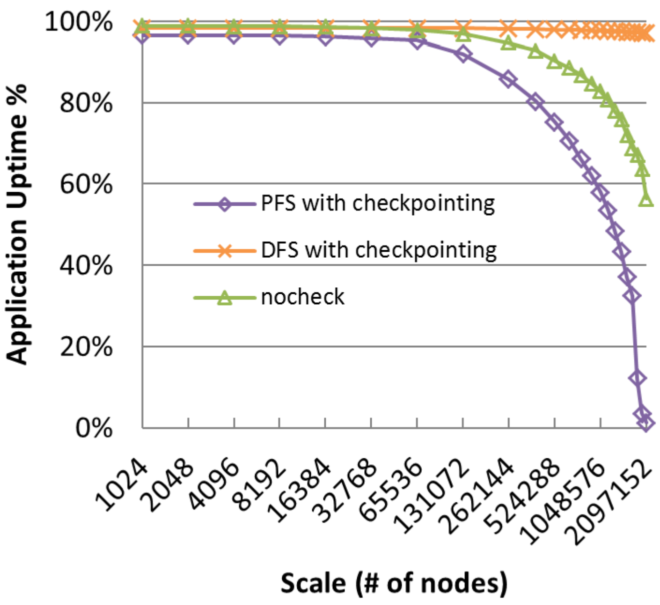
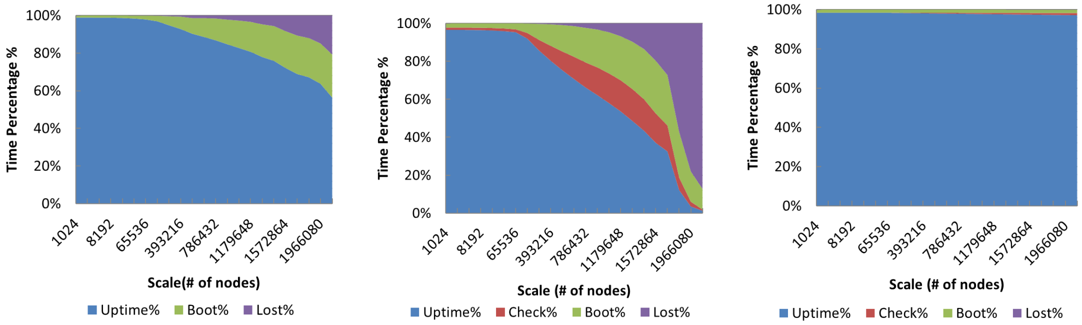


Figure 2.19. Application efficiency for Blue Gene/P jobs



(a) No checkpointing (b) PFS with checkpointing (c) DFS with checkpointing

Figure 2.20. Cost breakdown of checkpointing on Blue Gene/P

We show the cost breakdown of different filesystems in Figure 2.20. PFS has a significant portion of checkpointing overhead, booting time and lost-time at exascale ($\geq$ 1 million nodes), as shown in Figure 2.20(b). DFS, on the other hand, introduces negligible overhead ($< 5\%$) in Figure 2.20(c).

2.6 Summary

This chapter presents the simulation results of exascale systems with different system architectures. We developed the RXSim simulator to simulate checkpointing performance for exascale computing. RXSim justifies that distributed filesystems are more optimistic than state-of-the-art parallel filesystems for reliable exascale computers. In particular, we found that local persistent storage would be viable to leverage data locality in the context of traditional distributed filesystems. Our study shows that local storage would be one of the key points to succeed in maintaining the reliability for exascale computers. The results are coincident with the findings in [47], where a hybrid of local/global checkpointing mechanism was proposed for the projected exascale system.

CHAPTER 3
THE CO-LOCATION OF COMPUTE AND STORAGE

State-of-the-art yet decades old architecture of high performance computing (HPC) systems has its compute and storage resources separated. It has shown limits for today's data-intensive scientific applications, because every I/O needs to be transferred via the network between the compute and storage cliques. This chapter describes a distributed storage layer local to the compute nodes, which is responsible for most of the I/O operations and saves extreme amount of data movement between compute and storage resources. We have designed and implemented a system prototype of such architecture –the FusionFS distributed filesystem [234]– 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 in an IBM Blue Gene/P supercomputer, showing more than an order of magnitude performance improvement over other popular file systems such as GPFS [172], PVFS [30], and HDFS [178].

3.1 Background

The conventional architecture of high-performance computing (HPC) systems separates the compute and storage resources into two cliques (i.e., compute nodes and storage nodes), both of which are interconnected by a shared network infrastructure. This architecture is mainly a result 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. However, in the era of Big Data, scientific applications are becoming more and more data-intensive, requiring a greater degree of support from the storage subsystem [58].

While recent studies [107, 184] addressed the I/O bottleneck in the conventional architecture of HPC systems, this chapter is orthogonal to them by proposing a new storage architecture to co-locate the storage and compute resources. In particular, we envision a distributed storage system on compute nodes for applications to manipulate their intermediate results and checkpoints, rather than transferring data over the network. While co-location of storage and computation has been widely leveraged in data centers (e.g. Hadoop clusters), such architecture never exists in HPC systems. This work, to the best of our knowledge, for the first time demonstrates how to architect and engineer such a system, and reports how much, quantitatively, it could improve the I/O performance of real scientific applications.

The proposed architecture of co-locating compute and storage could raise concerns about jitters on compute nodes, since applications' computation and I/O share resources like 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 [48]. In fact, this resource-isolation strategy has been applied in production systems: the IBM Blue Gene/Q supercomputer (Mira [123]) 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 (for example, the popular distributed file system HDFS [178]); yet there exists little literature for building a distributed storage system particularly for HPC systems whose design principles are significantly different from data centers. HPC nodes are highly customized and tightly coupled with high throughput and low latency network (for example, InfiniBand), while data centers typically have commodity servers and inexpensive networks (for example, Ethernet). So storage systems designed for data centers are not optimized for the HPC machines, as we will discuss in more detail where HDFS shows poor performance on a typical HPC machine (Figure 3.12). 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:

(1) 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, as Welch and Noer [199] reported that 25% – 90% of all the 600 million files from 65 Panasas [134] installations are 64 KB or smaller. So the I/O performance is highly throttled by the metadata rate in addition to the data itself. Data centers, however, are not optimized for this type of workload. If we recall that HDFS [178] splits a large file into a series of default 64 MB chunks (128 MB recommended in most cases) for parallel processing, 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.

(2) File writes should be optimized for a distributed file system on HPC compute nodes. The fault tolerance of most 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 to roll back to the most recent correct checkpoint up on a failure. So file writes typically outnumber file reads in terms of both frequency and size in HPC systems, thus improving the write performance will significantly reduce the overall I/O cost. The fault tolerance of data centers, however, is not achieved through checkpointing its memory states, but the re-computation of affected data chunks that are replicated on multiple nodes.

3.2 Design Overview

As shown in Figure 3.1, FusionFS [234, 225] 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.

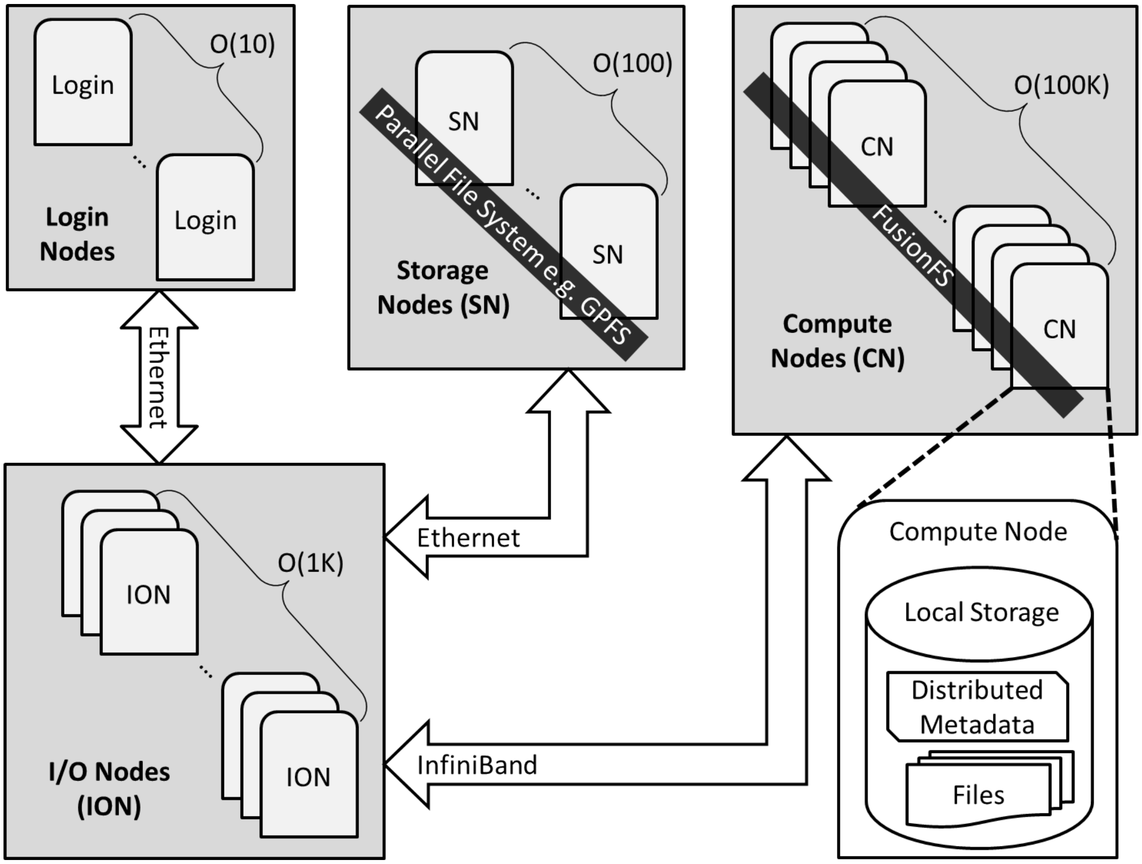


Figure 3.1. FusionFS deployment in a typical HPC system

FusionFS supports both the POSIX interface and a user library. The POSIX interface is implemented with the FUSE framework [59], so that legacy applications can run directly on FusionFS without modifications. Just like other user-level file systems (for example, PVFS [30]), 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 where to store the metadata and data, the mount point of the remote parallel file system (for example, Lustrue [173], GPFS [172], PVFS [30]), and the mount point of FusionFS where 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 (on the local compute nodes).

FUSE has been criticized for its performance overhead. In native UNIX-like file systems (for example, Ext4) there are 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 was reported in [162], showing that a Java implementation of a FUSE-based file system introduces about 60% overhead compared to 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 [226], 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 POSIX, for example *ffs_open()*, *ffs_close()*, *ffs_read()*, and *ffs_write()*. The downside of this approach is the lack of POSIX support, implying that the application might not be portable to other file systems and often needs modification and recompilation.

3.3 Metadata Management

3.3.1 Namespace. Clients have a coherent view of all the files in FusionFS no matter if the file is stored in the local node or a remote node. This global namespace is maintained by a distributed hash table (DHT [101, 102]), which disperses partial metadata on each compute node. As shown in Figure 3.2, in this example Node 1 and Node 2 only physically store 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 inquiry 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 it does not physically exist in any data structure, the global namespace does not need to 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.

3.3.2 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, there is a field *filelist* to record all the entries under this directory. This *filelist* field is particularly useful for providing an in-memory speed for directory read such as “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, Figure 3.3 shows the distributed hash table according to the example metadata shown in Figure 3.2. 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

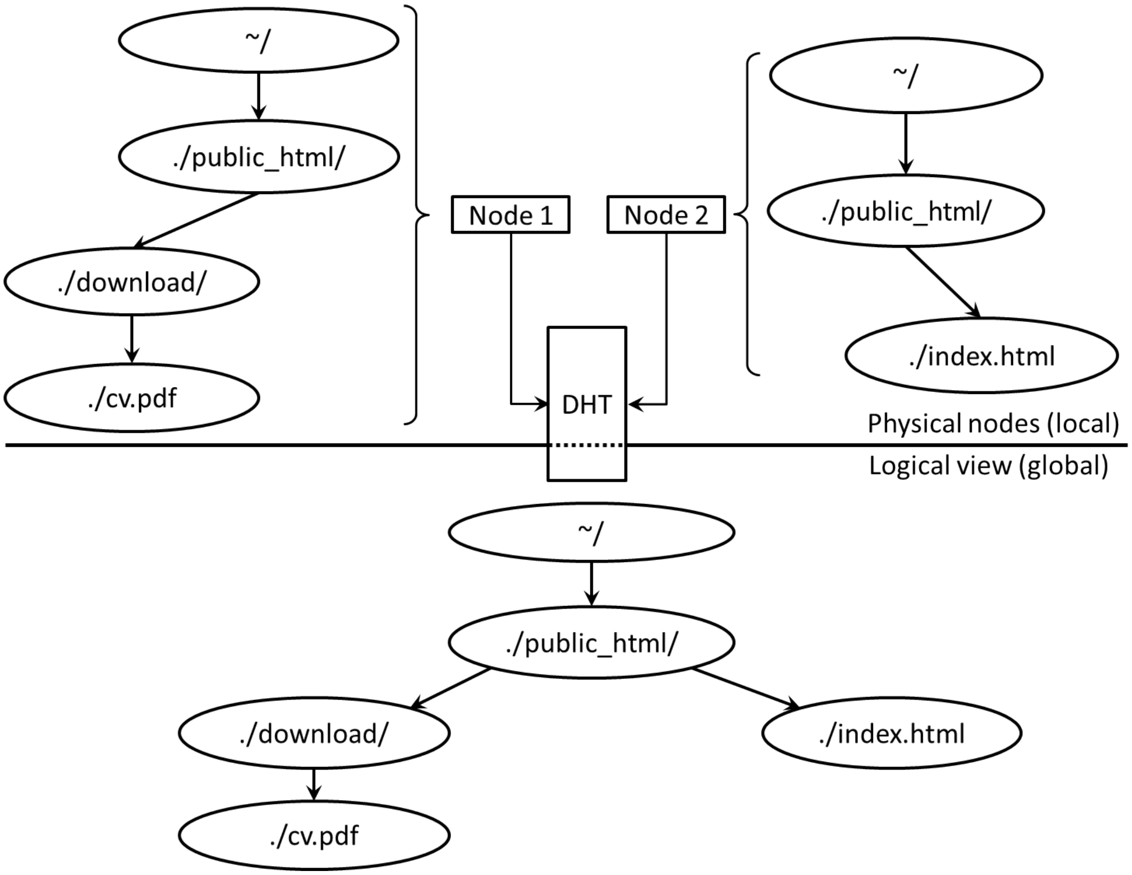


Figure 3.2. Metadata in the local nodes and the global namespace

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, such as the IP address of the node on which the file is stored. The value in the figure as a string delimited by semicolons is, in fact, only for clear representation. In implementation, the value is stored in a C structure. Upon a client request, this value structure is serialized by Google Protocol Buffers [153] 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. We will have more to say about how to parallelize the serialization in Chapter 8.

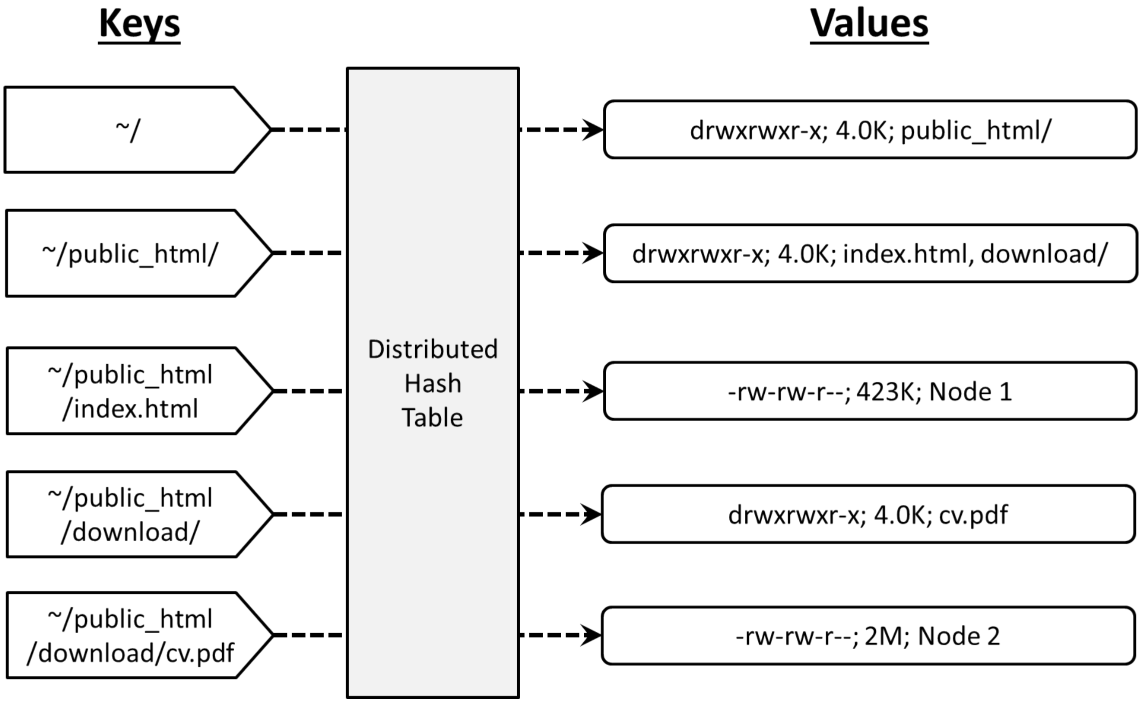


Figure 3.3. The global namespace abstracted by key-value pairs in a DHT

The metadata and data on a local node are completely decoupled: a regular file's location is independent of its metadata location. From Figure 3.2, we know the *index.html* metadata is stored on Node 2 and the *cv.pdf* metadata is on Node 1. However, it is perfectly fine for *index.html* to reside on Node 1, and for *cv.pdf* to reside on Node 2, as shown in Figure 3.3. Besides the conventional metadata information for regular files, there is a special flag in the value indicating if this file is being written. Specifically, any client who requests to write a file needs to sets this flag before opening the file and will not reset it until the file is closed. The atomic compare-swap operation supported by DHT [101] guarantees the file consistency for concurrent writes.

Another challenge on the metadata implementation is on large directories. 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 slow. 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 mitigate that, we implement an atomic append operation that asynchronously appends the incremental change to the value. This approach is similar to Google File System [63], where files are immutable and can only be appended. This gives us excellent concurrent metadata modification in large directories at the expense of potentially slowing down read operations.

3.3.3 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 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, a header [message_id, packet_id] is added to the message at the sender, and the message is restored by sorting the packet_id for each message at the recipient. This is efficiently done by a sorted map with message_id as the key mapping to a sorted set of the message's packets.

3.3.4 Persistence. The whole point 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 just 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, it is similar to the incremental checkpointing mechanism. This asynchronous flushing helps to sustain the high performance of the in-memory metadata operations.

3.3.5 Fault Tolerance. When a node fails, we need to restore the missing metadata and files on that node as soon as possible. The traditional method for data replication is to make a number of replicas of the primary copy. When the primary copy is lost, one of the replicas will be restored to become the new primary copy. This method has its advantages such as ease-of-use, less compute-intensive, when compared to the emerging erasure-coding mechanism [149, 167]. The main critique on replicas is, however, its low storage efficiency. For example, in Google file system [63] each primary copy has two replicas, which results in a low storage utilization $\frac{1}{1+2} = 33\%$.

For fault tolerance of metadata, we chose data replication over erasure coding based on the following observations. First, metadata size is typically much smaller than file data in orders of magnitude. Therefore, replicating the metadata impacts little to the overall space utilization of the entire system. Second, the computation overhead introduced by erasure coding can be hardly amortized by the reduced I/O time on transferring the encoded metadata. In essence, erasure coding is preferred when data is large and the time needed to encode and decode files would be offset by the benefit of sending less data to remote nodes (i.e., less network consumption). This is not the case for transferring metadata, where the primary metric is latency (rather than bandwidth).

3.3.6 Consistency. Since each primary metadata copy has replicas, the next questions is how make them consistent. Traditionally, there are two semantics to keep replicas consistent: (1) strong consistent – blocking until replicas are finished with updating; (2) weak consistent – return 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 (for example, last modification time), the first replica is strong consistent to the primary copy and the other replicas are updated asynchronously. By doing this, the metadata are strong consistent (in the average case) while the overhead is kept relatively low.

3.4 Data Movement Protocols

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

We have developed our own data transfer service Fusion Data Transfer (FDT) on top of UDP-based Data Transfer (UDT) [67]. 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 TCP.

3.4.2 File Open. Figure 3.4 shows the protocol when opening a file in FusionFS. Due to limited space, we assume the requested file is also on Node-j. Note that it is not necessarily Node-j who stores both the requested file and its metadata, as the metadata and data are decoupled on compute nodes.

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 [101]. The location information might be stored in another machine Node-j, so this procedure could involve a round trip of messages between

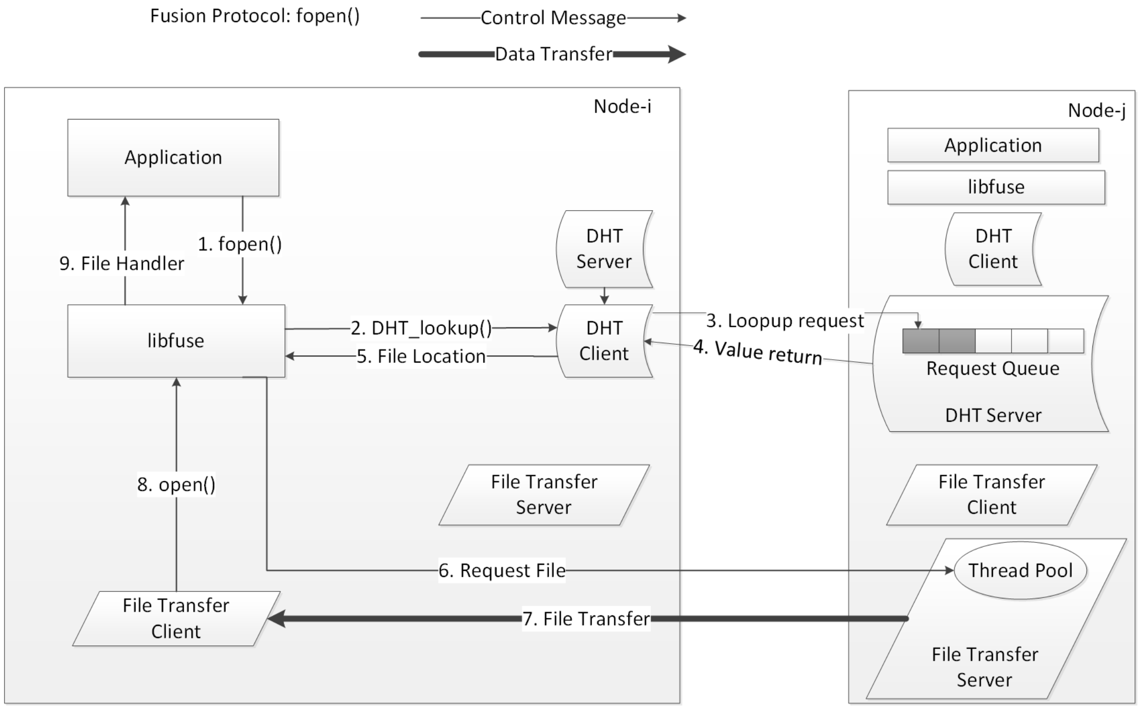


Figure 3.4. The protocol of file open in FusionFS

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 and finally step 9 returns the file handle to the application.

3.4.3 File Write. Before writing to a file, the process checks if the file is being accessed by another process. If so, an error number is returned to the caller. Otherwise the process can do one of the following two things. 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. 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 obviously optimal because file writes are associated with local I/O throughput and avoids the following two types of cost: (1) the procedure to determine 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 re-balance 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 saves the cost of transferring the file data over the network.

3.4.4 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, e.g. Swift [236]. 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 [158], 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, e.g. merge sort. In merge sort, the data need 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. FDT service is deployed on each compute node, and keeps listening to the incoming fetch and send requests.

3.4.5 File Close. Figure 3.5 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.

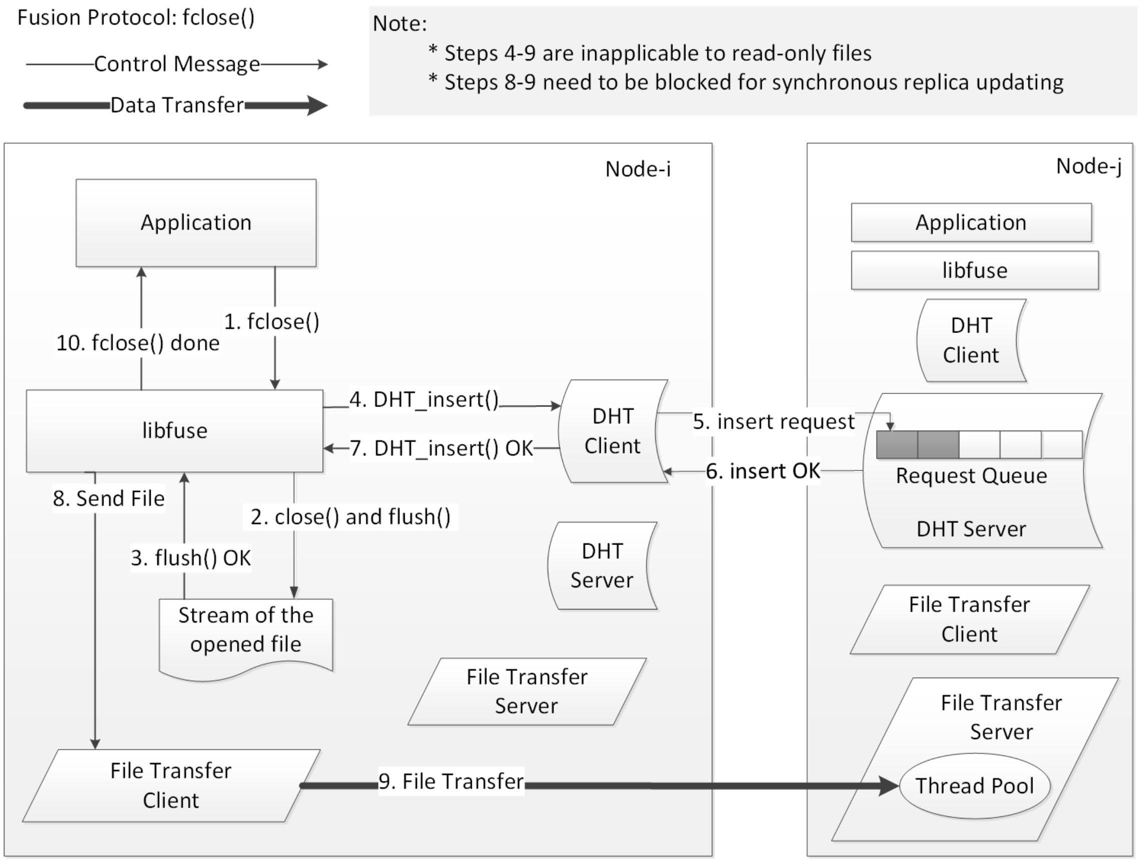


Figure 3.5. The protocol of file close in FusionFS

Again, just like Figure 3.4, the replica is not necessarily stored on the same node of its metadata (Node-j).

3.5 Experiment Results

While we indeed compare FusionFS to some open-source systems such as PVFS [30] (in Figure 3.8) and HDFS [178] (in Figure 3.12), 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 [188], 4 systems are IBM Blue Gene/Q systems which run GPFS [172] as the default file system. Therefore most large-scale experiments conducted in this section are carried out on Intrepid [79], a 40K-node IBM Blue Gene/P supercomputer whose default file system is also GPFS.

Intrepid serves as a test bed for FusionFS more as a demonstration of the scalability we plan to achieve in a hypothetical deployment with many compute nodes and node-local storage. Note that FusionFS is not a customized file system only for Intrepid, but an implementation for HPC compute nodes in general.

Each Intrepid compute node has quad core 850MHz PowerPC 450 processors and runs a light-weight Linux ZeptoOS [212] with 2 GB memory. A 7.6PB GPFS [172] 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 are repeated at least five times, or until results become stable (within 5% margin of error). The reported numbers are the average of all runs. Caching effect is carefully precluded by reading a file larger than the on-board memory before the measurement.

3.5.1 Metadata Rate. We expect that the metadata performance of FusionFS should be significantly higher than the remote GPFS on Intrepid, because FusionFS manipulates metadata in a completely distributed manner on compute nodes while 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, both FusionFS and GPFS create 10K empty files from each client on its own directory on Intrepid. That is, at 1024-nodes scale, we create 10M files over 1024 directories. We could have let all clients write on the same directory, but this workload would not take advantage of GPFS' multiple I/O nodes. That is, we want to optimize GPFS' performance when comparing it to FusionFS.

As shown in Figure 3.6, at 1024-nodes scale, FusionFS delivers nearly two orders of magnitude higher metadata rate over GPFS. FusionFS shows excellent scalability, with no sign of slowdown up to 1024-nodes. The gap between GPFS and FusionFS metadata performance would continue to grow, as eight nodes are enough to saturate the metadata servers of GPFS.

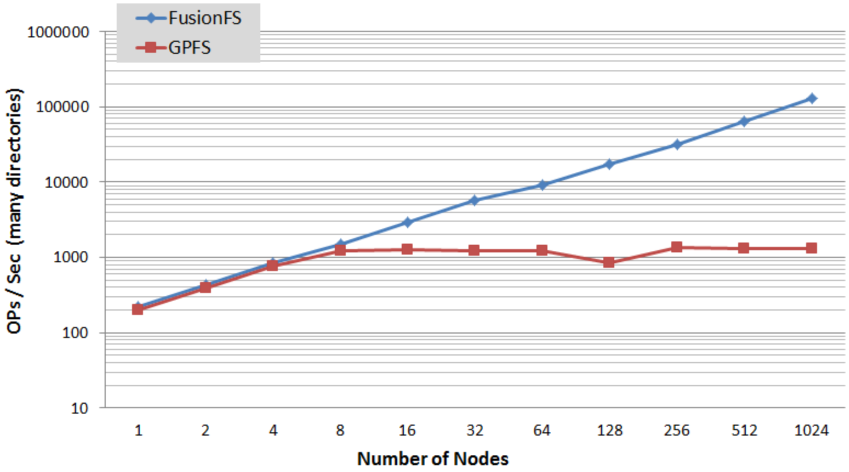


Figure 3.6. Metadata performance of FusionFS and GPFS on Intrepid (many directories)

Figure 3.7 shows the metadata throughput when multiple nodes create files in a single global directory. In this case, GPFS does not scale even with 2 nodes. FusionFS delivers scalable throughput with similar trends as in the many-directory case.

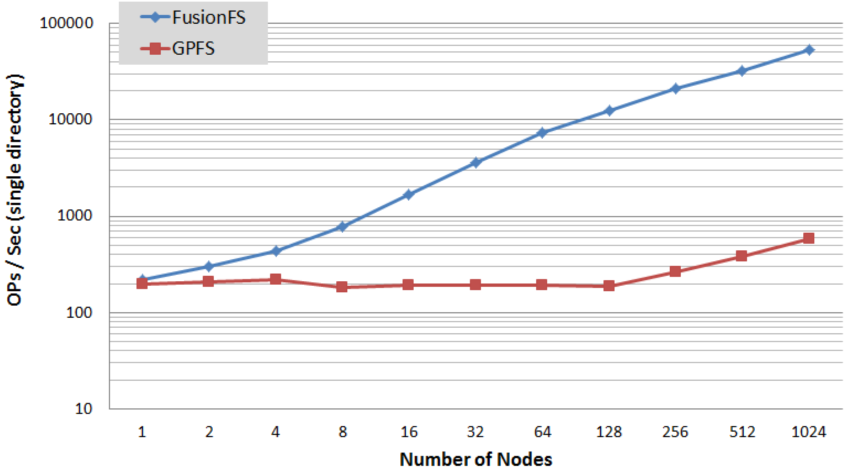


Figure 3.7. Metadata performance of FusionFS and GPFS on Intrepid (single directory)

One might overlook FusionFS' 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). First of all, that is the whole point why FusionFS is designed like this: 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 just like FusionFS?", we install PVFS [30] on Intrepid compute nodes with the 1-1-1 mapping between clients, metadata servers, and data servers just like FusionFS. We do not deploy GPFS on compute nodes because it is a proprietary system, and PVFS is open-sourced. The result is reported in Figure 3.8. 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, which justifies that the metadata optimization for the big directory (i.e. update → append) on FusionFS is highly effective. Unsurprisingly, 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 capability to handle higher concurrency.

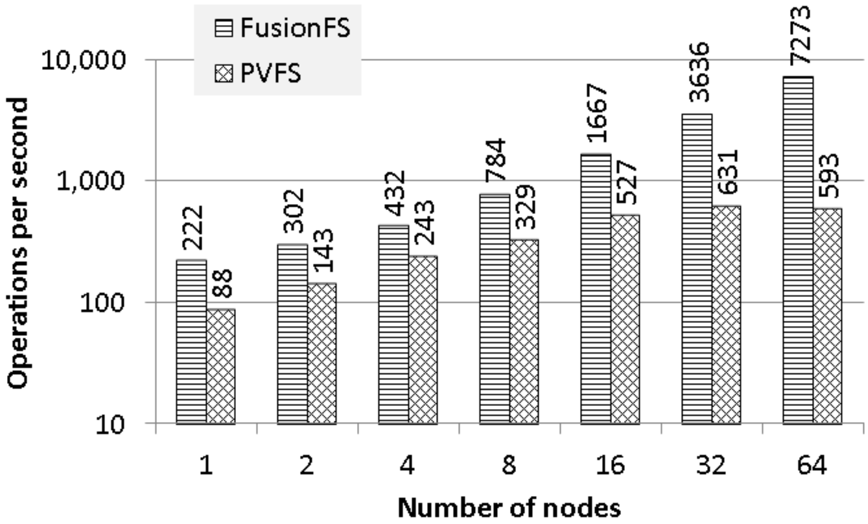


Figure 3.8. Metadata performance of FusionFS and PVFS on Intrepid (single directory)

3.5.2 I/O Throughput. Similarly to the metadata, we expect a significant improvement to the I/O throughput from FusionFS. Figure 3.9 shows the aggregate write throughput of FusionFS and GPFS on up to 1024-nodes of Intrepid. FusionFS shows almost linear scalability across all scales. GPFS scales at a 64-nodes step because every 64 compute nodes share one I/O node. Nevertheless, GPFS is still orders of magnitude slower than FusionFS at all scales. In particular, at 1024-nodes, FusionFS outperforms GPFS with a 57X higher throughput (113 GB/s vs. 2 GB/s).

Figure 3.10 shows FusionFS's scalability at extreme scales. The experiment is carried out on Intrepid on up to 16K-node each of which has a FusionFS mount point. FusionFS throughput shows about linear scalability: doubling the number of nodes yield doubled throughput. Specifically, we observe stable 2.5 TB/s throughput

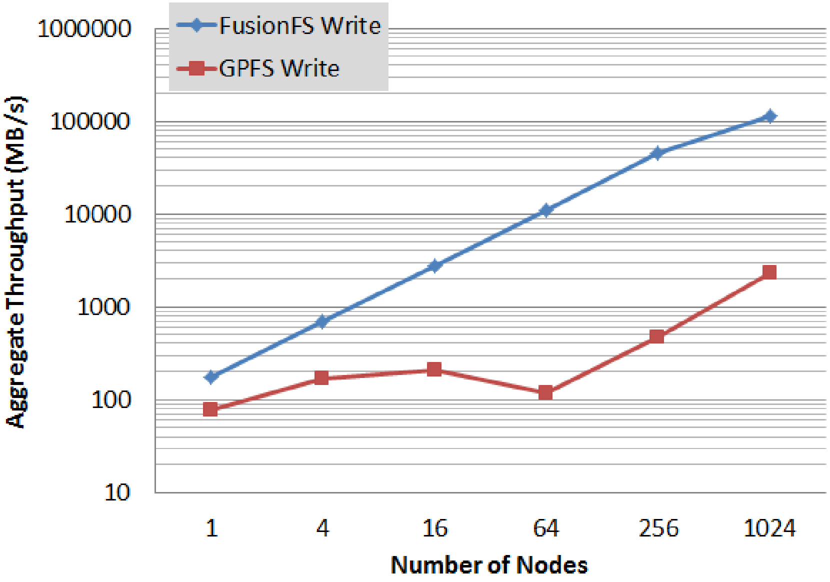


Figure 3.9. Write throughput of FusionFS and GPFS on Intrepid

(peak 2.64 TB/s) on 16K-nodes.

The main reason why FusionFS data write is faster is that the compute node only writes to its local storage. This is not true for data read though: it is possible that one node needs to transfer some remote data to its local disk. Thus, we are interested in two extreme scenarios (i.e. all-local read and all-remote read) that define the lower and upper bounds of read throughput. We measure 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.

Figure 3.11 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

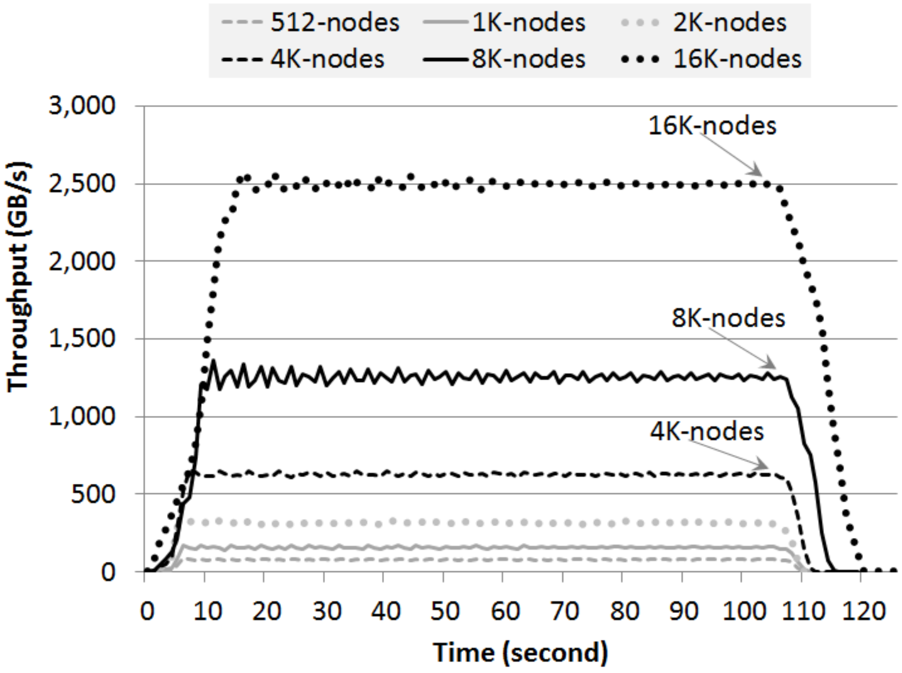


Figure 3.10. FusionFS scalability on Intrepid

considerably as the all-local case. The reason why all-remote reads still outperforms GPFS is, again, FusionFS' main concept of co-locating computation and data on the compute nodes: the bi-section 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).

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 (for example, Swift [236]) and job scheduling systems (for example, Falkon [160]) to implement the data-aware scheduling. Note that Falkon has already implemented a data-aware scheduler for the "data diffusion" storage system [160], a precursor to the FusionFS project that lacked distributed metadata management, hierarchical directory-based

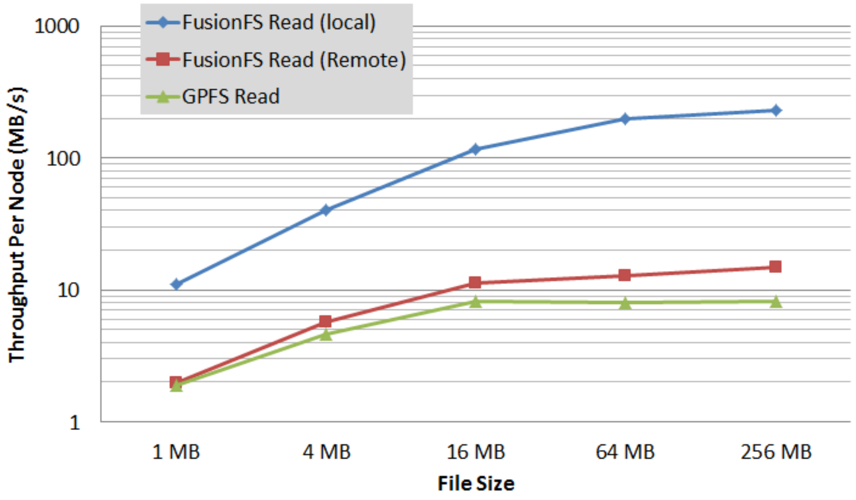


Figure 3.11. Read throughput of FusionFS and GPFS on Intrepid

namespace, and POSIX support.

It might be argued 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. First of all, that is the whole point of FusionFS: taking advantage of the fast interconnects across the compute nodes. Nevertheless, we want to emphasize that FusionFS' unique I/O strategy also plays a critical role in reaching the high and scalable throughput. So it would be a more fair game to compare FusionFS to other distributed file systems in the same hardware, architecture, and configuration. To show such a comparison, we deploy FusionFS and HDFS [178] on the Kodiak [93] cluster. We compare them on 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), 4GB RAM, and two 7200 rpm 1TB 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 to achieve the highest possible performance, and turn off the FUSE interface.

Figure 3.12 shows that the aggregate throughput of FusionFS outperforms HDFS by about an order of magnitude. FusionFS shows an excellent scalability, while HDFS starts to taper off at 256 nodes, mainly due to the weak write locality as data chunks (64 MB) need to be scattered out to multiple remote nodes.

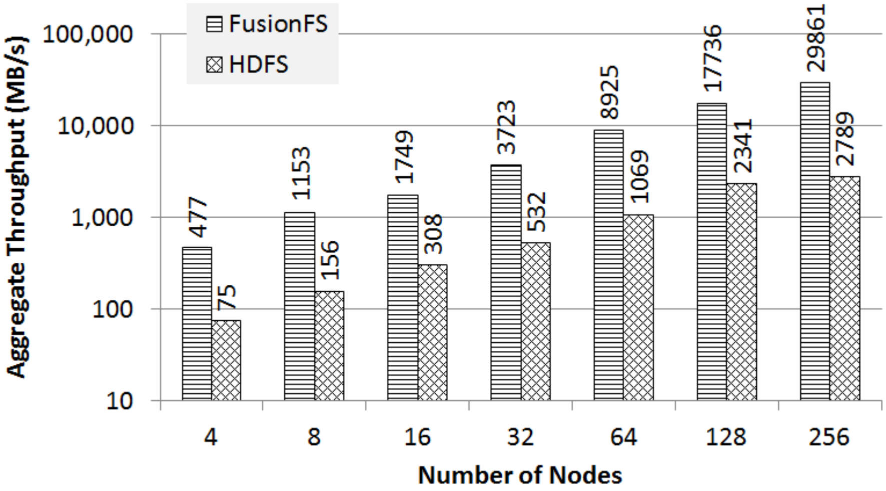


Figure 3.12. Throughput of FusionFS and HDFS on Kodiak

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

At last, we compared FusionFS with two other popular file systems on the cloud: S3FS [169] and Ceph [197]. Figure 3.13 shows the aggregate throughput of both FusionFS and S3FS on Amazon EC2 m3.medium instances. Obviously, S3FS is orders of magnitude slower than FusionFS. The reason is that every file operation on S3FS invokes a sequence of remote data transfer while FusionFS tries to deal with data on the local disk. FusionFS has shown highly strong scalability on HPC machines on up to 16K nodes resulting in an aggregate 2.5 TB/s throughput; we are now seeing a similar success on the cloud.

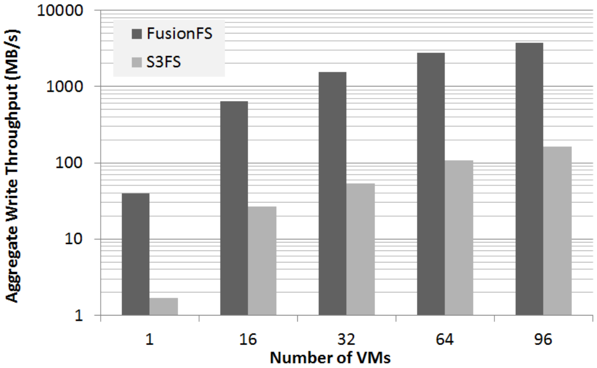


Figure 3.13. Write throughput of FusionFS and S3FS are compared.

According to FusionFS's performance, we envision the node-local storage is essential to achieve optimal write throughput. To further justify that, we conduct similar experiments with another node-local distributed file system Ceph [197] deployed on FermiCloud [54]. Figure 3.14 shows that the write throughput of Ceph is almost linearly scalable and the efficiency is always higher than 90%. The baseline number is relatively low (6.66 MB/s) because we only allow five writers in the experiment.

The scalable throughput of Ceph is attributed by its distributed metadata and data movement. The metadata in Ceph is organized in such a way that high availability and scalability are guaranteed. An extra ceph metadata server can be standby, ready to take over the duties of any failed metadata server that was active, thus eliminating the single point of failure on metadata. Moreover, Ceph can be configured with multiple metadata servers that split the directory tree into subtrees (and shards of a single busy directory), effectively balancing the load amongst all the active servers. Ceph use Object Storage Device (OSD) Daemons to handle the

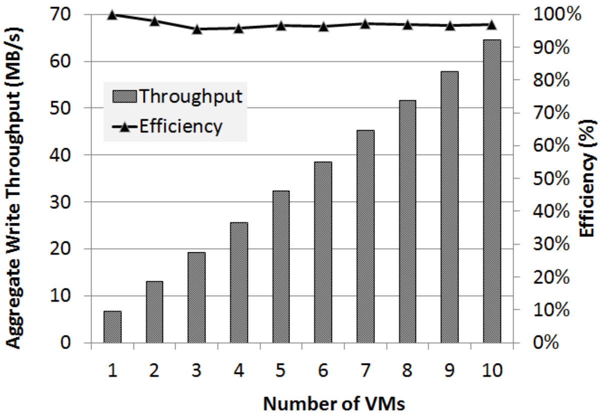


Figure 3.14. Scalable Write Throughput of Ceph.

read/write operations on the storage disks. Unlike traditional architectures, where clients talk to a centralized component (for example, a gateway, a broker), Ceph allows clients to interact with Ceph OSD Daemons directly [198]. This design prevents the single point of failure and improves the performance and scalability of Ceph's I/O throughput.

Nevertheless, a key difference exists between Ceph and FusionFS. In Ceph, a complicated algorithm (CRUSH) migrates data across multiple physical nodes to achieve load balance in nearly real-time; FusionFS always writes to the local storage and asynchronously calls a background process to balance the data. In other words, FusionFS trades real-time load balance for higher I/O performance, which is desirable in many scenarios.

Figure 3.15 shows the aggregate throughput of Ceph and FusionFS on Amazon EC2 cloud, both of which are deployed with FUSE and measured when writing/reading a distinct file of 4 GB on each node. The setup of both filesystems are the same: each node behaves as a data server, a metadata server, and a client. We observe both file systems have good scalability, but FusionFS is stronger: from 4 to 16 nodes the gap between Ceph and FusionFS is increased from $(2.0 + 3.1)/2 = 2.5\times$ to

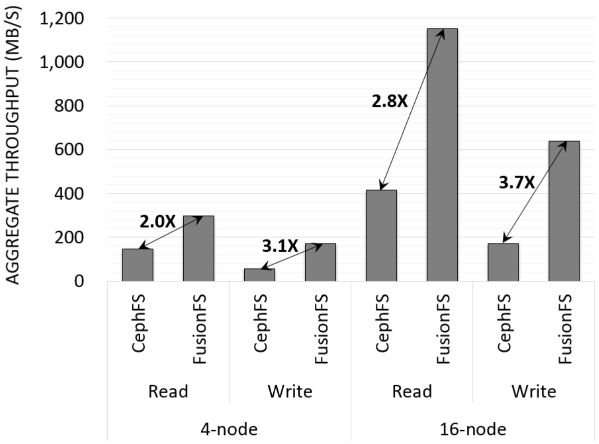


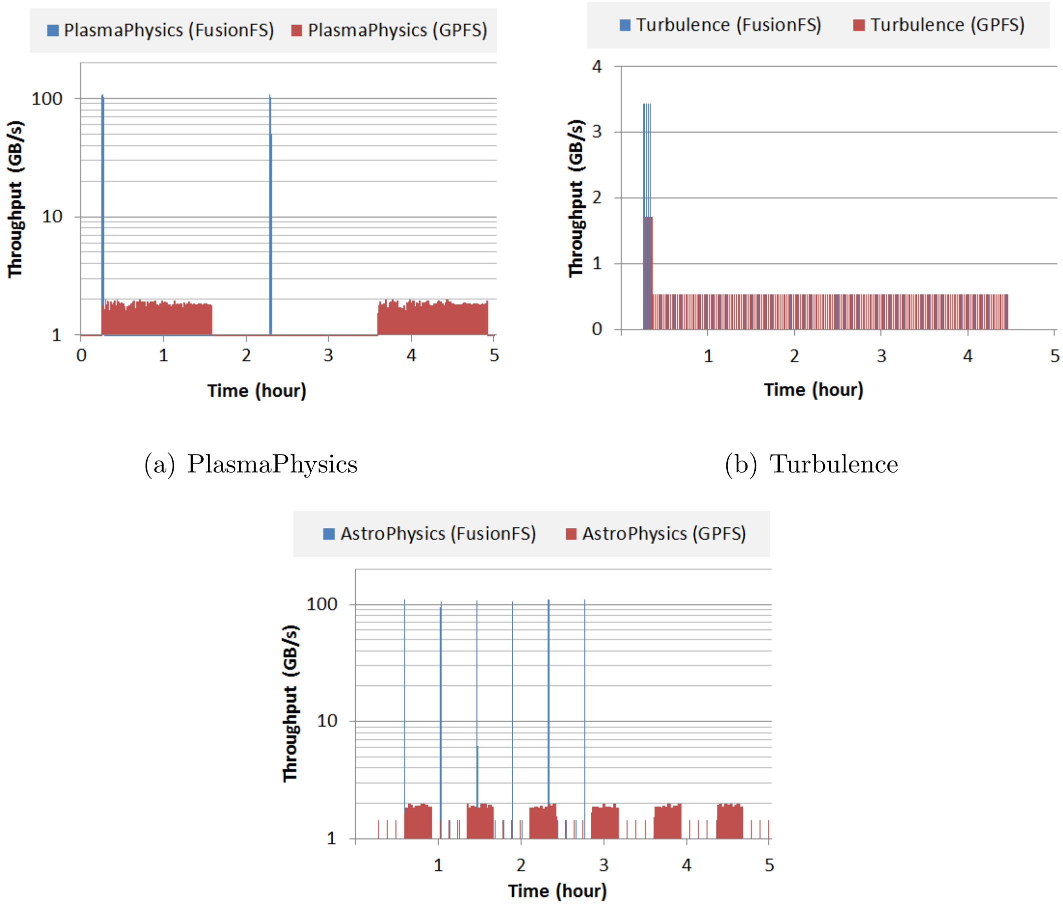
Figure 3.15. Ceph and FusionFS on Amazon EC2.

(2.8+3.7)/2 = 3.3×. Also note that the throughput of FusionFS is reaching the hardware limit: on 4 nodes it delivers aggregate 170 MB/s indicating 42.5 MB/s per node, and the raw bandwidth is 44.9 MB/s. That is, FusionFS achieves 42.5/44.9 = 94.7% efficiency.

3.5.3 Applications. We are interested in, quantitatively, how FusionFS helps to reduce the I/O cost for real applications. This section will evaluate four scientific applications on FusionFS all on Intrepid. The performance is mainly compared to Intrepid's default storage, the GPFS [172] parallel file system.

For the first three applications, we replay the top three write-intensive applications in December 2011 [107] 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' results comparable to GPFS for Turbulence and AstroPhysics, we modify both workloads to write to unique files as the exclusive chunks of the share file. Due to limited space, only the first five hours of these applications running on GPFS are considered.

Figure 3.16 shows the real-time I/O throughput of these workloads at 1024-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 of these workloads are completed in 5 hours in GPFS.



(a) PlasmaPhysics

(b) Turbulence

(c) AstroPhysics

Figure 3.16. Top three write-intensive applications on Intrepid

It is noteworthy that for both the PlasmaPhysics and AstroPhysics applications, the peak I/O rates for GPFS top at around 2GB/s while for FusionFS they reach over 100GB/s. This increase in I/O performance accelerates the applications 2.1X times (PlasmaPhysics) and 1.6X times (AstroPhysics). The reason why Turbulence does not benefit much from FusionFS is that, there are not many consecutive I/O operations in this application 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, e.g. ParallelBLAST [117]. In ParallelBLAST, the entire database (4GB) 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 [217]. We carried out a weak scaling experiment of ParallelBLAST with 4GB 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 produces an overall data I/O of 541GB over 16192 files for every 64-nodes. Figure 3.17 shows the workload in all three stages and the number of accessed files from 1 node to 1024 nodes. In our experiment of 1024-node scale, the total I/O is about 9TB applied to over 250,000 files.

As shown in Figure 3.18, there is 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 1024-nodes. At 1-node scale, the GPFS kernel module is more effective in accessing an idle parallel file system. In FusionFS' case, the 1-node scale result involves the user-level FUSE module, which apparently

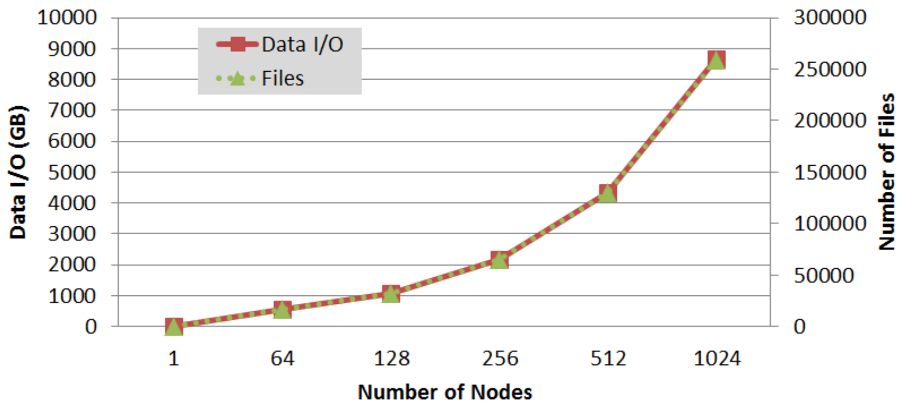


Figure 3.17. The workload over three stages of BLAST

causes BLAST to run 1.4X slower on FusionFS. However, beyond the corner-case of 1-node, FusionFS significantly outperforms GPFS. In particular, on 1024-nodes BLAST requires 1,073 seconds to complete all three stages on FusionFS, and it needs 32,440 seconds to complete the same workload on GPFS.

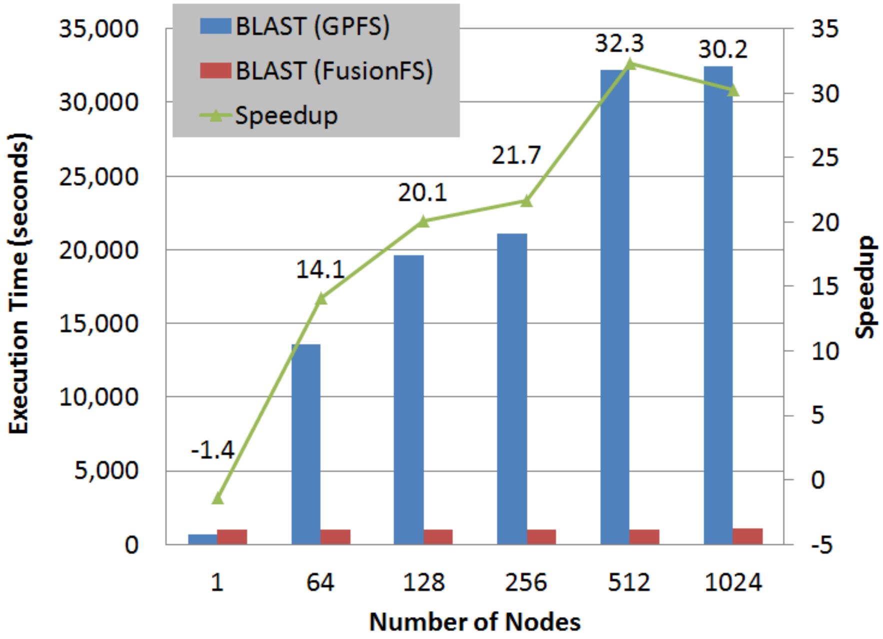


Figure 3.18. BLAST execution time on Intrepid

Based on the speedup of FusionFS and GPFS at different scales, we show the efficiency of both systems in Figure 3.19. FusionFS still keeps a high efficiency (i.e. 64%) at 1024-nodes scale, where GPFS falls below 5% at 64-nodes and beyond. For this application, GPFS is an extremely poor choice, as it cannot handle the concurrency generated by the application beyond 64-nodes. Recall that this GPFS file system has total 128 storage nodes in Intrepid, and is configured to support concurrent accessing from 40K compute nodes, yet it exhibits little scaling from as small as 64-nodes.

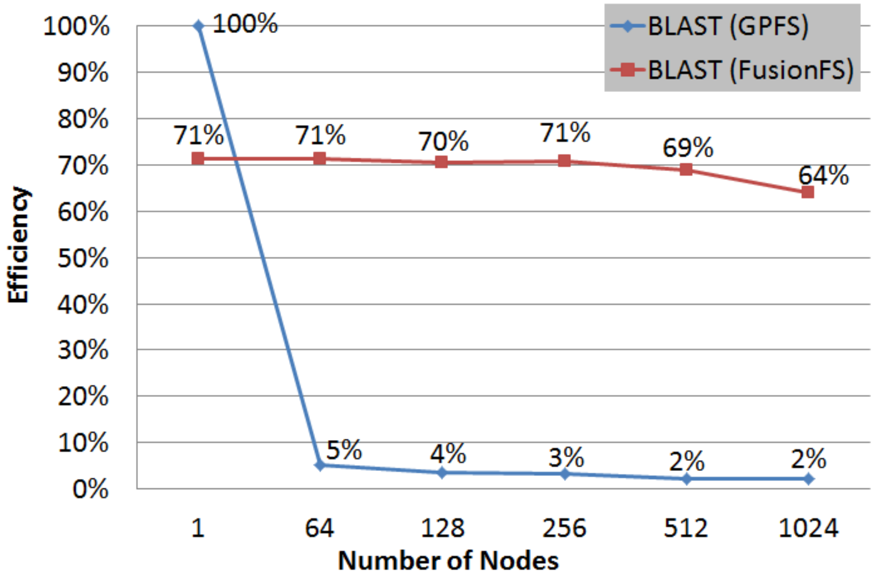


Figure 3.19. BLAST I/O efficiency on Intrepid

Lastly, we measure the overall throughput of data I/O at different scales as generated by the BLAST application in Figure 3.20. FusionFS has an excellent scalability reaching over 8GB/s, and GPFS is saturated at 0.27GB/s from 64 nodes and beyond.

3.6 Limitation

Load balance. The I/O strategy of FusionFS is optimized for file write, and does not take load balance into account. Recall that every client tries to write

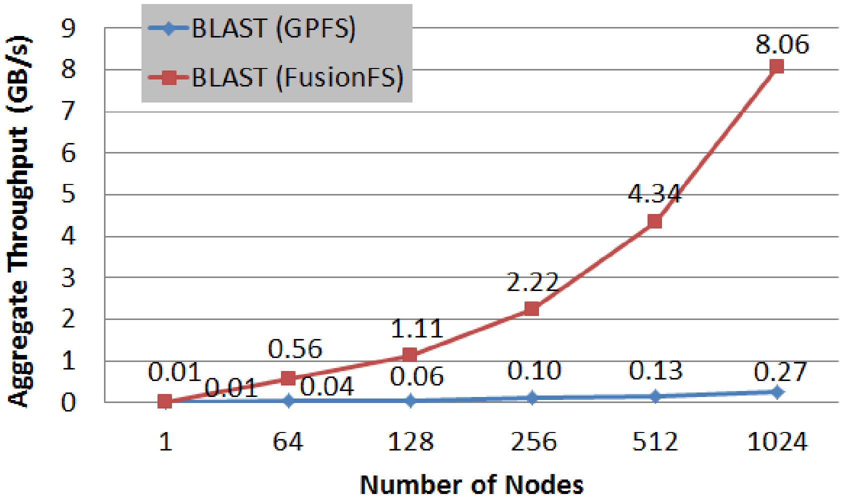


Figure 3.20. BLAST I/O throughput on Intrepid

memory to the local disk, even for those files whose original location is a remote node. FusionFS just updates the metadata instead of sending back the updated file to its original location. So for those workloads where a subset of active nodes conduct significantly more file writes than other nodes, we expect more data to reside on these active nodes, which puts the system in an imbalance state of storage consumption. This issue could be addressed by an asynchronous re-balance procedure running in the background, or by a load-aware task scheduler that steals tasks from the active nodes to the more idle ones.

N-to-1 Write. The current FusionFS implementation works on the file granularity. This implies that multiple nodes concurrently writing to different portions of the same file would cause a non-deterministic result. While we could manually split the file into sub-files of exclusive chunks to be accessed concurrently, a more elegant solution would be for FusionFS to provide an API or runtime option to turn on this chunk-level manipulation.

3.7 Summary

This chapter introduces 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 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' superiority over other popular storage systems for scientific applications.

CHAPTER 4
DISTRIBUTED CACHING

One of the bottlenecks of distributed file systems deals with mechanical hard drives (HDD). Although solid-state drives (SSD) have been around since the 1990's, HDDs are still dominant due to large capacity and relatively low cost. Hybrid hard drives with a small built-in SSD cache does not meet the need of a large variety of workloads.

This section proposes a middleware that manages the underlying heterogeneous storage devices in order to allow distributed file systems to leverage the SSD performance while leveraging the capacity of HDD. We design and implement a user-level filesystem, HyCache [226], that can offer SSD-like performance at a cost similar to a HDD. We show how HyCache can be used to improve performance in distributed file systems, such as the Hadoop HDFS.

We then extend HyCache to HyCache+ [222] –a cooperative cache on the compute nodes–, which allows I/O to effectively leverage the high bi-section bandwidth of the high-speed interconnect of massively parallel high-end computing systems. HyCache+ provides the POSIX interface to end users with the memory-class I/O throughput and latency, and transparently swap the cached data with the existing slow-speed but high-capacity networked attached storage. HyCache+ has the potential to achieve both high performance and low-cost large capacity, the best of both worlds. To further improve the caching performance from the perspective of the global storage system, we propose a 2-phase mechanism to cache the hot data for parallel applications, called 2-Layer Scheduling (2LS), which minimizes the file size to be transferred between compute nodes and heuristically replaces files in the cache.

4.1 HyCache: Local Caching with Memory-Class Storage

HyCache is designed to manage heterogeneous storage devices for distributed filesystems. HyCache provides standard POSIX interfaces through FUSE [59] and works completely in the user space. We show that in the context of filesystems, the overhead of user-level APIs (i.e. *libfuse*) is negligible with multithread support on SSD, and with appropriate tuning can even outperform the kernel-level implementation. The user-space feature of HyCache allows non-privileged users to specify the SSD cache size, an invaluable feature in making HyCache more versatile and flexible to support a much wider array of workloads. Furthermore, distributed or parallel filesystems can leverage HyCache without any modifications through its POSIX interface. This is critical in many cases where the end users are not allowed to modify the kernel of HPC systems. Figure 4.1 shows the conceptual view of the storage hierarchy with HyCache. Instead of being mounted directly on the native file systems (e.g. Linux Ext4), distributed file systems are deployed on top of HyCache on all data nodes.

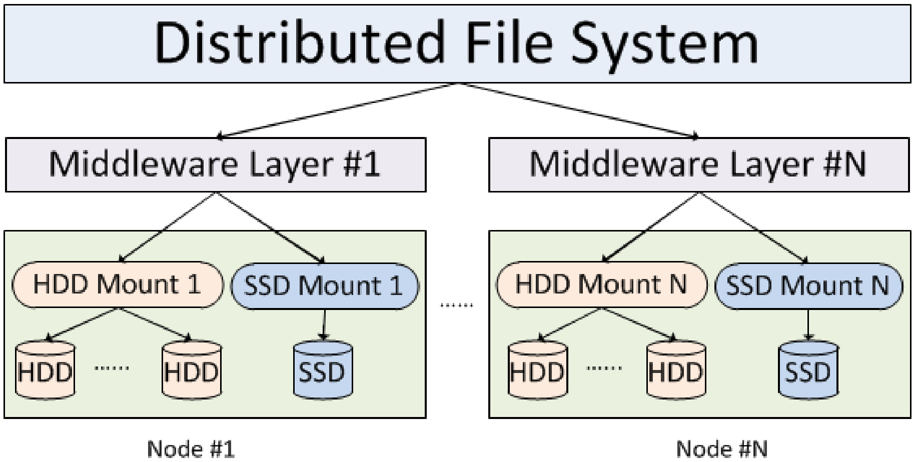


Figure 4.1. The storage hierarchy with a middleware between distributed file systems and local file systems.

4.1.1 Design Overview. Figure 4.2 shows a bird's view of HyCache as a middleware between distributed file systems and local storages. At the highest level there are three logical components: request handler, file dispatcher and data manipulator. Request handler interacts with distributed file systems and passes the requests to the file dispatcher. File dispatcher takes file requests from request handler and decides where and how to fetch the data based on some replacement algorithm. Data manipulator manipulates data between two access points of fast- and regular-speed devices, respectively.

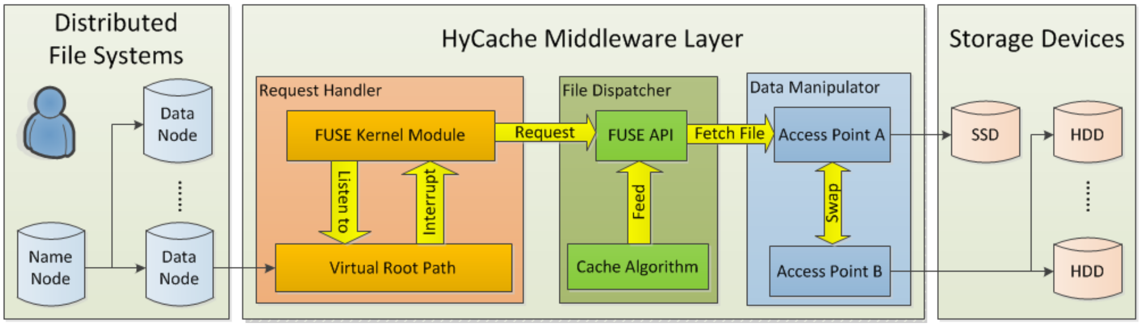


Figure 4.2. Three major components in HyCache architecture: Request Handler, File Dispatcher and Data Manipulator.

The request handler is the first component of the whole system that interacts with distributed file systems. HyCache virtual mount point can be any directory in a UNIX-like system as long as end users have sufficient permissions on that directory. This mount point is monitored by the FUSE kernel module, so any POSIX file operations on this mount point is passed to the FUSE kernel module. Then the FUSE kernel module will import the FUSE library and try to transfer the request to FUSE API in the file dispatcher.

File dispatcher is the core component of HyCache, as it redirects user-provided POSIX requests into customized handlers of file manipulations. FUSE only provides POSIX interfaces and file dispatcher is exactly the place where these interfaces are implemented, e.g. some of the most important file operations like *open()*, *read()* and *write()*, etc. File dispatcher manages the file locations and determines with which hard drive a particular file should be dealing. Some replacement policies, i.e. cache algorithms, need to be provided to guide the File Dispatcher.

Cache algorithms are optimizing instructions that a computer program can follow to manage a cache of information stored on the computer. When the cache is full, the algorithm must choose which items to discard to make room for the new ones. In case of HyCache, cache algorithm determines which file(s) in SSD are swapped to HDD when the SSD space is intensive. Different cache algorithms have been extensively studied in the past decades. There is no one single algorithm that suppresses others in all scenarios. We have implemented LRU (Least Recently Used) and LFU (Least Frequently Used) [180] in HyCache and the users are free to plug in their own algorithms for swapping files.

Data manipulator manipulates data between two logical access points: one for fast speed access, i.e. SSDs and the other is for regular access e.g. HDDs. An access point is not necessarily a mount point of a device in the local operating system, but a logical view of any combination of these mount points. In the simplest case, Access point A could be the SSD mount point whereas access Point B is set to the HDD mount point. Access point A is always the preferred point for any data request as long as it has enough space based on some user defined criteria. Data need to be swapped back and forth between A and B once the space usage in A exceeds some threshold. For simplicity we only show Access point A and B in the figure, however there is nothing architecturally that prohibits us from leveraging more than two levels of access points.

4.1.2 User Interface. The HyCache mount point itself is not only a single local directory but a virtual entry point of two mount points for SSD partition and HDD partition, respectively. Figure 4.3 shows how to mount HyCache in a UNIX-like system. Assuming HyCache would be mounted on a local directory called *hycache_mount*, and another local directory (e.g. *hycache_root*) has been created and has at least two subdirectories: the mount point of the SSD partition and the mount point of the HDD partition, users can simply execute *./hycache <root> <mount>* where *hycache* is the executable for HyCache, *root* is the physical directory and *mount* is the virtual directory.

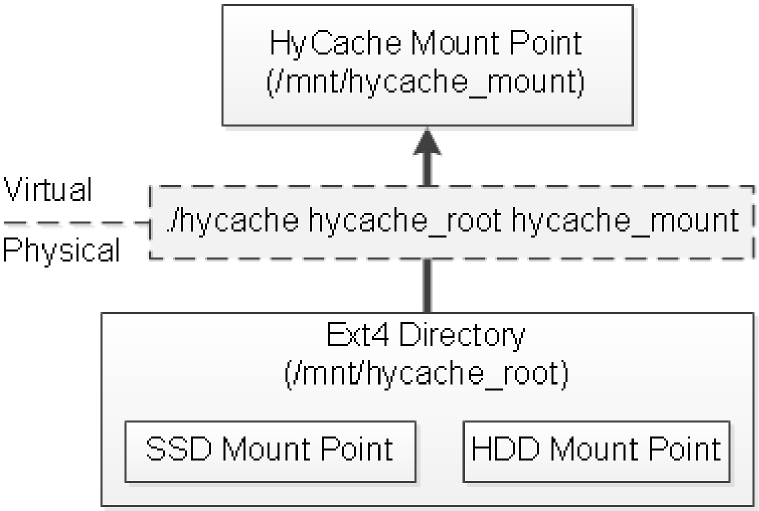


Figure 4.3. How to mount HyCache in a UNIX-like machine.

4.1.3 Strong Consistency. We keep only one single copy of any file at any time to achieve strong consistency. For manipulating files across multiple storage devices we use symbolic links to track file locations. Another possibility is to adopt hash tables. In this initial release we preferred symbolic links to hash tables for two reasons. First, symbolic link itself is persistent, which means that we do not need to worry about the cost of swapping data between memory and hard disk. Second, symbolic link is directly supported by UNIX-like systems and FUSE framework.

HyCache is implemented for manipulating data at the file level rather than the block level because it is the job of the upper-level distributed filesystem to chop the big files (e.g. > 1TB). For example in HDFS an arbitrarily large file will typically be chopped up in 64MB chunks on each data node. Thus HyCache only needs to deal with these relatively small data blocks of 64MB that can be perfectly fit in a mainstream SSD device.

4.1.4 Single Namespace. Figure 4.4 shows a typical scenario of file mappings when the space of SSD cache is intensive so some file(s) needs to be swapped into the HDD. End users only see virtual files in HyCache mount point (i.e. *hycache_mount*) and every single file in the virtual directory is mapped to the underlying SSD physical directory. SSD only has a limited space so when the usage is beyond some threshold then HyCache will move some file(s) from SSD to HDD and only keep symbolic link(s) to the swapped files. The replacement policy, e.g. LRU or LFU, determines when and how to do the swapping.

We illustrate how a file is opened as an example. Algorithm 4.1 describes how HyCache updates SSD cache when end users open files. The first thing is to check if the requested file is physically in HDD in Line 1. If so the system needs to reserve enough space in SSD for the requested file. This is done in a loop from Line 2 to Line 5 where the stale files are moved from SSD to HDD and the cache queue is updated. Then the symbolic link of the requested file is removed and the physical one is moved from HDD to SSD in Line 6 and Line 7. We also need to update the cache queue in Line 8 and Line 10 for two scenarios, respectively. Finally the file is opened in Line 12.

Another important file operation in HyCache that is worth mentioning is file removal. We explain how HyCache removes a file in Algorithm 4.2. Line 4 and Line 5 are standard instructions used in file removal: update the cache queue and remove the file. Lines 1-3 check if the file to be removed is actually stored in HDD. If so, this regular file needs to be removed as well.

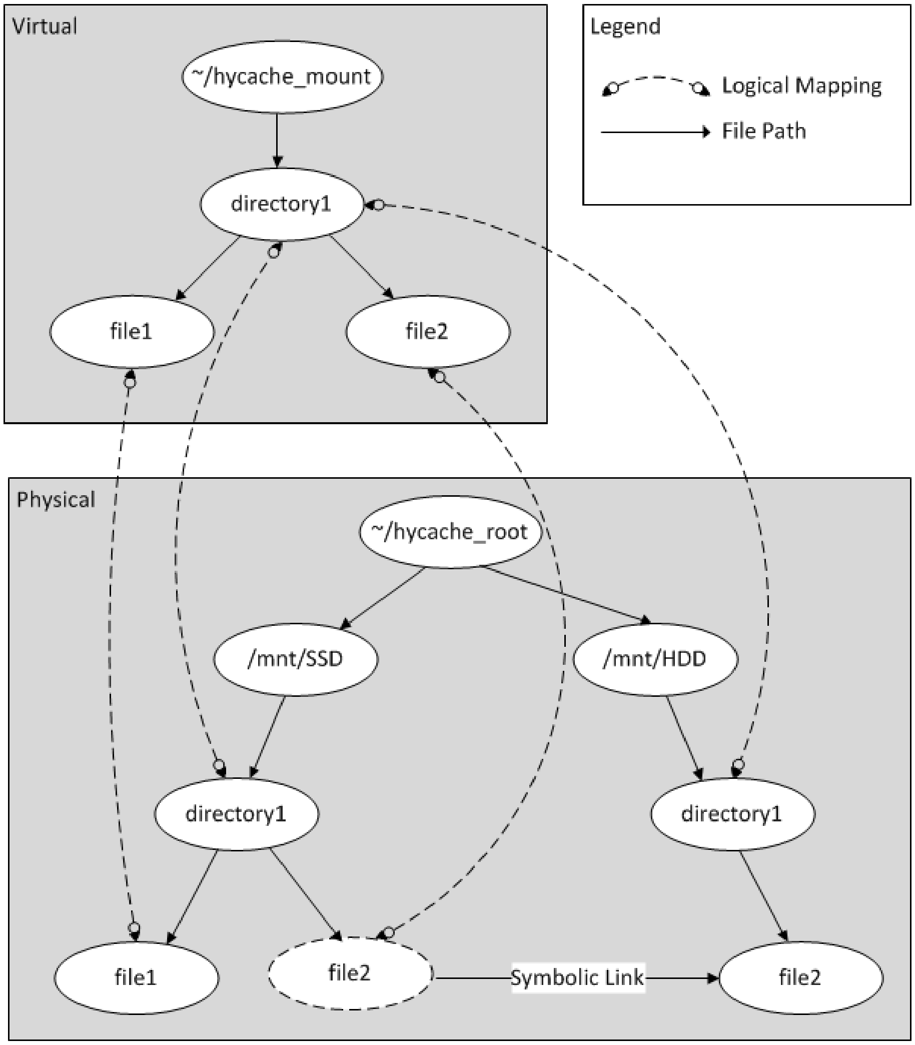


Figure 4.4. File movement in HyCache

Other POSIX implementations share the similar idea to Algorithm 4.1 and Algorithm 4.2: manipulate files in SSD and HDD back and forth to make users feel they are working on a single file system, e.g. *rename()*, which is to rename a file in Algorithm 4.3. If the file to be renamed is a symbolic in SSD, the corresponding file in HDD needs to be renamed as shown in Line 2. Then the symbolic link in SSD is outdated and needs to be updated in Lines 3-4. On the other hand if the file to be renamed is only stored in SSD then the renaming occurs only in SSD and the cache queue, as shown in Lines 6-7. In either case the position of the newly accessed file F'

Algorithm 4.1 Open a file in HyCache

Input: F is the file requested by the end user; Q is the cache queue used for the replacement policy; SSD is the mount point of SSD drive; HDD is the mount point of HDD drive

Output: F is appropriately opened

```
1:  if F is a symbolic link in SSD then
2:      while SSD space is intensive and Q is not empty do
3:          move some file(s) from SSD to HDD
4:          remove these files from the Q
5:      end while
6:      remove symbolic link of F in SSD
7:      move F from HDD to SSD
8:      insert F to Q
9:  else
10:     adjust the position of F in Q
11: end if
12: open F in SSD
```

in the cache queue needs to be updated in Line 9.

4.1.5 Caching Algorithms. HyCache provides two built-in cache algorithms: LRU and LFU. End users are free to plug in other cache algorithms depending on their data patterns and application characteristics. As shown in Algorithms 4.1, all the implementations are independent of specific cache algorithms. LRU is one of the most widely used cache algorithms in computer systems. It is also the default cache algorithm used in HyCache. LFU is an alternative to facilitate the SSD cache if the access frequency is of more interests. In case all files are only accessed once (or for equal times), LFU is essentially the same as LRU, i.e. the file that is least recently

Algorithm 4.2 Remove a file in HyCache

```
Input: F is the file requested by the end user for removal; Q is the cache queue used
    for the replacement policy; SSD is the mount point of SSD drive; HDD is the
    mount point of HDD drive
Output: F is appropriately removed
if F is a symbolic link in SSD then
    remove F from HDD
end if
remove F from Q
remove F from SSD
```

Algorithm 4.3 Rename a file in HyCache

```
Input: F is the file requested by the end user to rename; F' is the new file name;
    Q is the queue used for the replacement policy; SSD is the mount point of SSD
    drive; HDD is the mount point of HDD drive
Output: F is renamed to F'
if F is a symbolic link in SSD then
    rename F to F' in HDD
    remove F in SSD
    create the symbolic link F' in SSD
else
    rename F to F' in SSD
    rename F to F' in Q
end if
update F' position in Q
```

used would be swapped to HDD if SSD space becomes intensive. We implement LRU and LFU with the standard C library <*search.h*> instead of importing any third-party libraries for queue-handling utilities. This header supports doubly-linked list with only two operation: *insque()* for insertion and *remque()* for removal. We implement all other utilities from scratch e.g. check the queue length, search for a particular element in the queue, etc. Each element of LRU and LFU queues stores some metadata of a particular file like filename, access time, number of access (only useful for LFU though), etc.

Figure 4.5 illustrates how LRU is implemented for HyCache. A new file is always created on SSD. This is possible because HyCache ensures the SSD partition has enough space for next file operation after current file operation. For example after editing a file, the system checks if the usage of SSD has hit the threshold of being considered as “SSD space is intensive”. Users can define this value by their own, for example 90% of the entire SSD. When the new file has been created on SSD it is also inserted in to the tail of LRU queue. On the other hand, if the SSD space is intensive we need to keep swapping the heads of LRU queue into HDD until the SSD usage is below the threshold. Both cases are pretty standard queue operations as shown in the top part of Figure 4.5. If a file already in the LRU queue gets accessed then we need to update its position in the LRU queue to reflect the new time stamp of this file. In particular, as shown in the bottom part of Figure 4.5, the newly accessed file needs to be removed from the queue and re-inserted into the tail.

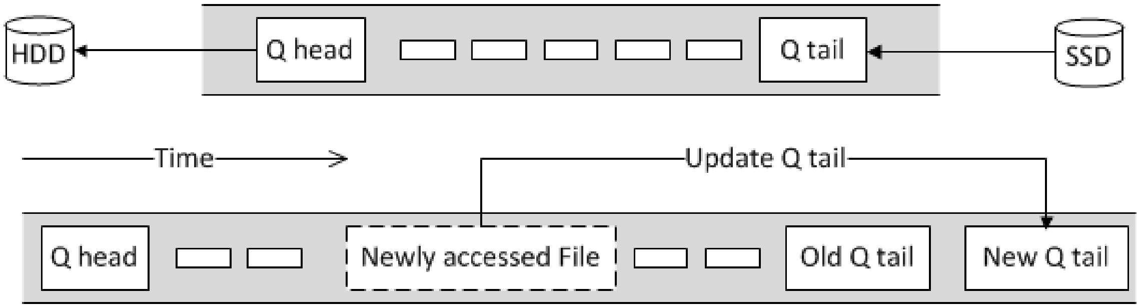


Figure 4.5. LRU queue in HyCache.

LFU is implemented in a similar way as LRU with a little more work. In

LFU, the position of a file in the queue is determined by two criteria: frequency and timestamp. LFU first checks the access frequency of the file. The more frequently this file has been touched, the closer it will be positioned to the queue tail. If there are multiple files with the same frequency, for this particular set of files LRU will be applied, i.e. based on timestamp.

4.1.6 Multithread Support.

HyCache fully supports multithreading to leverage the many-core architecture in most high performance computers. Users have the option to disable this feature to run applications in the single-thread mode. Even though there are cases where multithreading does not help and only introduces overheads by switching contexts, by default multithreading is enabled in HyCache because in most cases this would improve the overall performance by keeping the CPU busy. We will see in the evaluation section how the aggregate throughput is significantly elevated with the help of concurrency.

4.2 HyCache+: Cooperative Caching among Many Nodes

This section presents a distributed storage middleware, called HyCache+, right on the compute nodes, which allows I/O to effectively leverage the high bi-section bandwidth of the high-speed interconnect of massively parallel high-end computing systems. HyCache+ provides the POSIX interface to end users with the memory-class I/O throughput and latency, and transparently swap the cached data with the existing slow-speed but high-capacity networked attached storage. HyCache+ has the potential to achieve both high performance and low-cost large capacity, the best of both worlds. To further improve the caching performance from the perspective of the global storage system, we propose a 2-phase mechanism to cache the hot data for parallel applications, called 2-Layer Scheduling (2LS), which minimizes the file size to be transferred between compute nodes and heuristically replaces files in the cache.

Figure 4.6 shows the design overview of HyCache+, on an oversimplified 2-node cluster. A job scheduler deploys a job on a specific machine, *Node 1* in this example. The hot files are accessed from the local cache if possible, and can potentially be replaced by the cold files in the remote parallel file systems according to the caching algorithm, e.g. LRU or Algorithm 4.5 that will be presented later. The hot files could be migrated between compute nodes with extremely high throughput and low latency, since most HPC systems are deployed with high-speed interconnect between compute nodes in order to meet the needs of large scale compute-intensive applications.

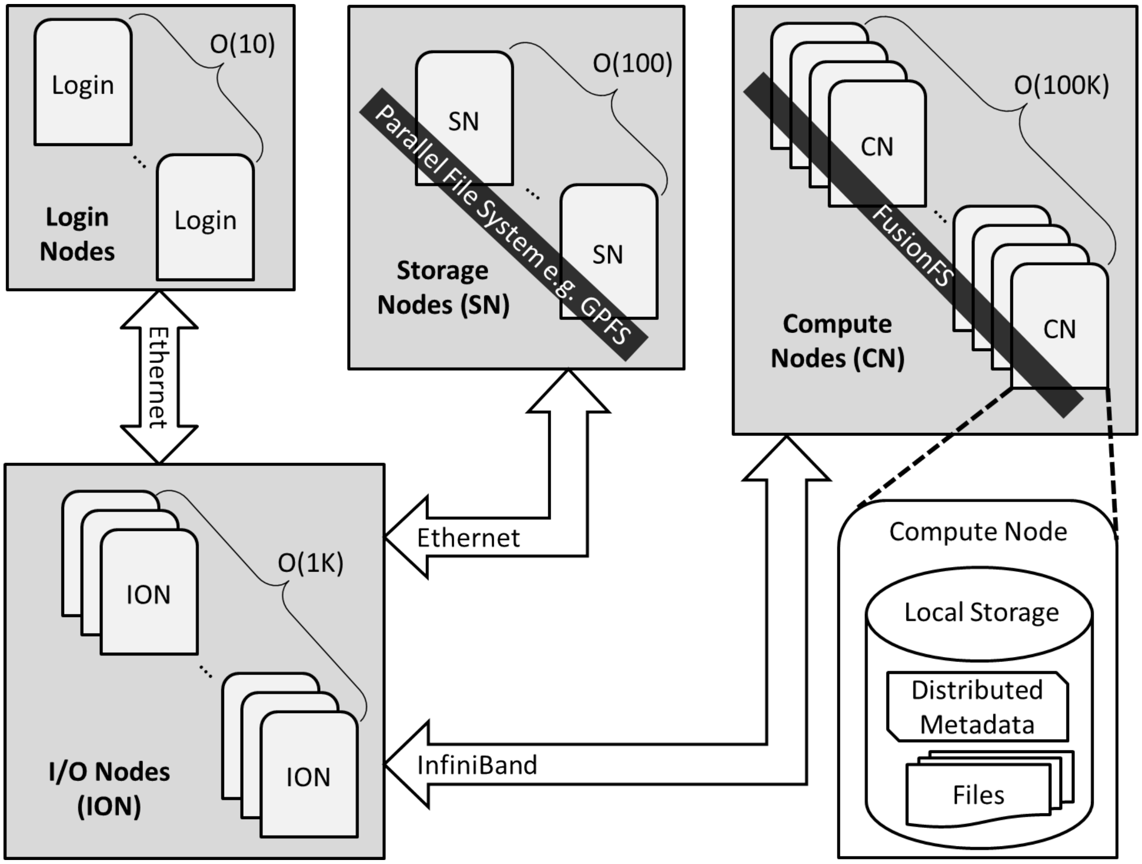


Figure 4.6. HyCache+ design overview

4.2.1 User Interface. One of our design goals is to provide complete transparency of the underlying storage heterogeneity to the users. By transparency, we mean that users are agnostic about which files are stored on which underlying storage types, or which physical nodes. This transparency is achieved by a global view of metadata of all the dispersed files.

In general, it is critical for a distributed/parallel file system to support POSIX for HPC applications, since POSIX is one of the most widely used standard. For legacy reasons, most HPC applications assume that the underlying file system supports POSIX. For the sake of backward compatibility, POSIX should be supported if at all possible. HyCache+ leverages the FUSE framework [59] to support POSIX.

The FUSE kernel module has been officially merged into the Linux kernel tree since kernel version 2.6.14. FUSE provides 35 interfaces to fully comply with POSIX file operations. Some of these are called more frequently e.g. some essential file operations like *open()*, *read()*, *write()* and *unlink()*, whereas others might be less popular or even remain optional in some Linux distributions like *getxattr()* and *setxattr()* which are to get and set extra file attributes, respectively.

FUSE has been criticized for its efficiency on traditional HDD-based file systems. In native UNIX file systems (e.g. Ext4) there are only two context switches between the caller in user space and the system call in kernel spaces. However for a FUSE-based file system, context needs to be switched four times: two switches between the caller and the virtual file system; and another two between libfuse and FUSE. We will show that this overhead, at least in HPC systems when multi-threading is turned on, is insignificant.

HyCache+ is deployed as a user level mount point in accordance with other user level file systems. The mount point itself is a virtual entry point that talks to the local cache and remote parallel file system. For example (see Figure 4.7), HyCache+ could be mounted on a local directory called */mnt/hycacheplus/*, while two other physical directories */dev/ssd/* and */mnt/gpfs/*) are for the local cache and the remote parallel file system, respectively.

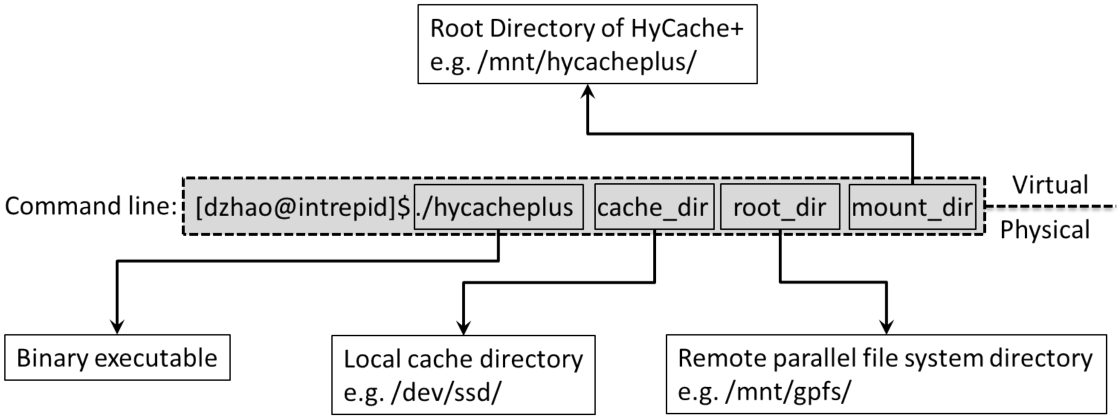


Figure 4.7. HyCache+ mountpoint

4.2.2 Job Scheduling. The variables to be used in the discussion are summarized in Table 4.1. If the application needs to access a file F_i on a remote machine, the overhead on transferring F_i is $Size(F_i)$.

Table 4.1. Variables of Global Scheduling

Variable	Type	Meaning
M	Set	Machines of the cluster
A	Set	Applications to be run
F	Set	All files
F^k	Set	Files referenced by $A_k \in A$
$P_{i,j}$	Int	$F_i \in F$ placed on $M_j \in M$
$Q_{i,j}$	Int	$A_i \in A$ scheduled on $M_j \in M$

We formalize the problem as to find the matrix Q (i.e. scheduling which job on which machine) that minimizes the overall network cost of running all $|A|$ jobs on

$|M|$ machines. That is to solve the objective function

$$\arg\min_{Q} \sum_{A_k \in A} \sum_{M_l \in M} \sum_{F_i \in F^k} \sum_{M_j \in M} Size(F_i) \cdot P_{i,j} \cdot Q_{k,l},$$

subject to

$$\sum_{M_j \in M} P_{i,j} = 1, \forall F_i \in F,$$

$$\sum_{M_j \in M} Q_{i,j} = 1, \forall A_i \in A,$$

$$P_{i,j}, Q_{i,j} \in \{0,1\}, \forall i, j.$$

The first constraint guarantees that a file could be placed on exact one node. Similarly, the second constraints guarantees that a job could be scheduled on exact one node. Note that both constraints could be generalized by replacing 1 with other constants if needed, for example in distributed file systems [178] a file could have multiple replicas for high reliability. The last constraint says that both matrices should only store binary values to guarantee the first and the second constraints.

The algorithm to find the machine for a job to achieve the minimal network cost is given in Algorithm 4.4. The input is the job index x, and it returns the machine index y. It loops on each machine (Line 4), calculates the cost of moving all the referenced files to this machine (Lines 5 - 9), and updates the minimal cost if needed (Lines 10 - 13).

The correctness of Algorithm 4.4 is due to the fact that the data locality is known as a priori input. That is, P is a given argument to the execution of all jobs. Otherwise Line 7 would not work appropriately. The per-job networking overhead is obviously minimal. Since jobs are assumed independent, the overall overhead of all jobs is also minimal.

The complexity of Algorithm 4.4 is $O(|M| \cdot |F_x|)$, by observing the two loops

Algorithm 4.4 Global Schedule

Input: The x^{th} job to be scheduled

Output: The y^{th} machine where the x^{th} job should be scheduled

```
function GlobalSchedule($x$)
    $MinCost \leftarrow \infty$
    $y \leftarrow$ NULL
    for $M_i \in M$ do
        $Cost \leftarrow 0$
        for $F_j \in F^x$ do
            Find $M_k$ such that $P_{j,k} = 1$
            $Cost \leftarrow Cost + Size(F_j)$
        end for
        if $Cost < MinCost$ then
            $MinCost \leftarrow Cost$
            $y \leftarrow i$
        end if
    end for
    return $y$
end function
```

on Line 4 and Line 6, respectively. Note that we could achieve an $O(1)$ cost for Line 7 by retrieving the metadata of file j.

4.2.3 Heuristic Caching. *Problem Statement.* The problem of finding optimal caching on multiple-disk is proved to be NP-hard [9]. A simpler problem on a single-disk setup has a polynomial solution [3], which is, unfortunately, too complex to be applied in real applications. An approximation algorithm was proposed in [28] with the restriction that each file size should be the same, which limits its use in practice. In fact, at small scale (e.g. each node has $O(10)$ files to access), a brute-force solution with dynamic programming is viable, with the same idea of the classical problem of traveling salesman problem (TSP) [17] with exponential time complexity. However, in real applications the number of accessed files could be as large as 10,000, which makes the dynamic programming approach feasible. Therefore we propose a heuristic algorithm of $O(n \lg n)$ (n is the number of distinct files on the local node) for each job, which is efficient enough for arbitrarily large number of files in practice, especially when compared to the I/O time of the disk.

Assumptions. We assume a queue of jobs, and their requested files are known on each node in a given period of time, which could be calculated from the scheduling results and the metadata information. This assumption is based on our observation of many workflow systems [236, 235], which implicitly make a similar assumption: users are familiar with the applications they are to run and they are able to specify the task dependency (often times automatically based on the high-level parallel workflow description). Note that the referenced files are only for the jobs deployed on this node, because there is no need to cache the files that will be accessed by the jobs deployed on remote nodes.

Notations and Definitions The access pattern of a job is represented by a sequence $R = (r_1, r_2, \ldots, r_m)$, where each r_i indicates one access to a particular file.

Note that the files referenced by different r_i's are possibly the same, and could be on the cache, or the disk. We use $File(r_i)$ to indicate the file object which r_i references to. The size of the referenced file by r_i is denoted by $Size(File(r_i))$. The *cost* is defined as the to-be-evicted file size multiplied by its access frequency after the current processing position in the reference sequence. The *gain* is defined as the to-be-cached file size multiplied by its access frequency after the current fetch position in the reference sequence. Since cache throughput is typically orders of magnitude higher than disks (i.e. O(10GB/s) vs. O(100MB/s)), in our analysis we ignore the time of transferring data between the processor and the cache. Similarly, when the file is swapped between cache and disks, only the disk throughput is counted. The cache size on the local node is denoted by C, and the current set of files in the cache is denoted by S. Our goal is to minimize the total I/O cost of the disk by determining whether the accessed files should be placed in the cache.

There are 3 rules to be followed in the proposed caching algorithms.

Rule 1. Every fetch should bring into the cache the very *next* file in the reference sequence if it is not yet in the cache.

Rule 2. Never fetch a file to the cache if the total cost of the to-be-evicted files is greater than the gain of fetching this file.

The first 2 rules specify which file to be fetched and when to do the fetch, and say nothing about evicting files. Rule 3 speaks about what files to be evicted and when to do the eviction.

Rule 3. Every fetch should discard the files in the increasing order of their cost until there is enough space for the newly fetched file. If the cache has enough space for the new file, no eviction is needed.

We elucidate the above 3 rules with a concrete example. Assume we have a random reference sequence $R = (r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9)$. Let $File(r_1) = F_1$, $File(r_2) = F_2$, $File(r_3) = F_3$, $File(r_4) = F_4$, $File(r_5) = F_3$, $File(r_6) = F_1$, $File(r_7) = F_2$, $File(r_8) = F_4$, $File(r_9) = F_3$, and $Size(F_1) = 20$, $Size(F_2) = 40$, $Size(F_3) = 9$, $Size(F_4) = 40$. Let the cache capacity be 100. According to *Rule 1*, the first three files to be fetched to cache are (F_1, F_2, F_3). Then we need to decide if we want to fetch F_4. Let $Cost(F_i)$ be the cost of evicting F_i. Then we have $Cost(F_1) = 20 \times 1 = 20$, $Cost(F_2) = 40 \times 1 = 40$, and $Cost(F_3) = 9 \times 2 = 18$. According to *Rule 3*, we sort the costs in the increasing order (F_3, F_1, F_2). Then we evict the files in the sorted list, until there is enough room for the newly fetched file F_4 of size 40. In this case, we only need to evict F_3, so that the free cache space is $100 - 20 - 40 = 40$, just big enough for F_4. Before replacing F_3 by F_4, *Rule 2* is referred to ensure that the cost is smaller than the gain, which is true in this case by observing that the gain of prefetching F_4 is 40, larger than $Cost(F_3) = 18$.

The caching procedure is presented in Algorithm 4.5, which is called when the i^{th} reference is accessed and $File(r_{i+1})$ is not in the cache. If $File(r_{i+1})$ is already in the cache, then it is trivial to keep processing the next reference, which is not explicitly mentioned in the algorithm. $File(r_{i+1})$ will not be cached if it is accessed only once (Line 2). Subroutine $GetFilesToDiscard()$ tries to find a set of files to be discarded in order to make more room to (possibly) accommodate the newly fetched file in the cache (Line 3). Based on the decision made by Algorithm 4.5, $File(r_{i+1})$ could possibly replace the files in D in the cache (Line 4 - 7). $File(r_{i+1})$ is finally read into the processor from the cache or from the disk, depending on whether $File(r_{i+1})$ is already fetched to the cache (Line 6).

The time complexity is as follows. Line 2 only takes $O(1)$ since it can be precomputed using dynamic programming in advance. $GetFilesToDiscard()$ takes $O(n \lg n)$ that will be explained when discussing Algorithm 4.6. Thus the overall time

Algorithm 4.5 Fetch a file to cache or processor

Input: i is the reference index being processed

```
procedure FETCH($i$)
    if $\{r_j | File(r_j) = File(r_{i+1}) \wedge j > i + 1\} \neq \emptyset$ then
        $flag, D \leftarrow GetFilesToDiscard(i, i + 1)$
        if $flag = successful$ then
            Evict $D$ out of the cache
            Fetch $File(r_{i+1})$ to the cache
        end if
    end if
    Access $File(r_{i+1})$ (either from the cache or the disk)
end procedure
```

complexity of Algorithm 4.5 is $O(n \lg n)$.

The $GetFilesToDiscard()$ subroutine (Algorithm 4.6) first checks if the summation of current cache usage and the to-be-fetched file size is within the limit of cache. If so, then there is nothing to be discarded (Line 2 - 4). We sort the files by their increasing order of cost at Line 9, because we hope to evict out the file of the smallest cost. Then for each file in the cache, Lines 11 - 18 check if the gain of prefetching the file outweighs the associated cost. If the new cache usage is still within the limit, then we have successfully found the right swap (Lines 19 - 21).

We will show that the time complexity of Algorithm 4.6 is $O(n \lg n)$. Line 5 takes $O(1)$ to get the total number of occurrences of the referenced file. Line 9 takes $O(n \lg n)$ to sort, and Lines 10 - 22 take $O(n)$ because there would be no more than n files in the cache (Line 10) and Line 11 takes $O(1)$ to collect the file occurrences. Both Line 5 and Line 11 only need $O(1)$ because we can precompute those values by

Algorithm 4.6 Get set of files to be discarded

Input: i is the reference index being processed; j is the reference index to be (possibly) fetched to cache

Output: $successful$ – $File(r_j)$ will be fetched to the cache and D will be evicted; $failed$ – $File(r_j)$ will not be fetched to the cache

```
function GetFilesToDiscard($i, j$)
    if $Size(S) + Size(File(r_j)) \leq C$ then
        return $successful$, $\emptyset$
    end if
    $num \leftarrow$ Number of occurrences of $File(r_j)$ from $j + 1$
    $gain \leftarrow num \cdot Size(File(r_j))$
    $cost \leftarrow 0$
    $D \leftarrow \emptyset$
    Sort the files in $S$ in the increasing order of the cost
    for $F \in S$ do
        $tot \leftarrow$ Number of references of $F$ from $i + 1$
        $cost \leftarrow cost + tot \cdot Size(F)$
        if $cost < gain$ then
            $D \leftarrow D \cup \{F\}$
        else
            $D \leftarrow \emptyset$
            return $failed$,$D$
        end if
        if $Size(S \setminus D) + Size(File(r_j)) \leq C$ then
            break
        end if
    end for
    return $successful$, $D$
end function
```

dynamic programming in advance. Thus the total time complexity is $O(n \lg n)$.

4.3 Experiment Results

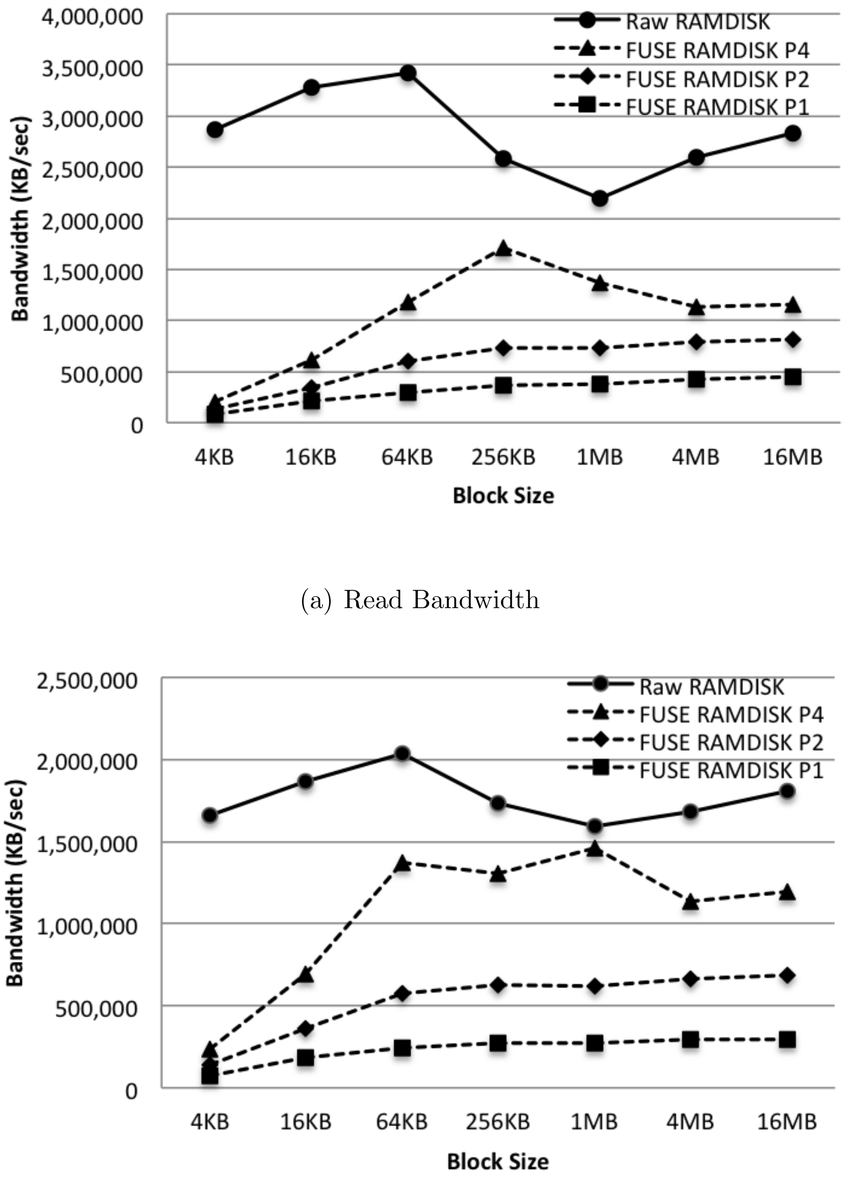
Single-node experiments of HyCache are carried out on a system comprised of an AMD Phenom II X6 1100T Processor (6 cores at 3.3 GHz) and 16 GB RAM. The spinning disk is Seagate Barracuda 1 TB. The SSD is OCZ RevoDrive2 100 GB. The HHD is Seagate Momentus XT 500 GB (with 4 GB built-in SSD cache). The operating system is 64-bit Fedora 16 with Linux kernel version 3.3.1. The native file system is Ext4 with default configurations (i.e. *mkfs.ext4 /dev/device*). For the experiments on Hadoop the testbed is a 32-node cluster, each of which has two Quad-Core AMD Opteron 2.3 GHz processors with 8 GB memory. The SSD and HDD are the same as in the single node workstation.

We have tested the functionality and performance of HyCache in four experiments. The first two are benchmarks with synthetic data to test the raw bandwidth of HyCache. In particular, these benchmarks can be further categorized into micro-benchmarks and macro-benchmarks. Micro-benchmarks are used to measure the performance of some particular file operations and their raw bandwidths. Macro-benchmarks, on the other hand, are focused on application-level performance of a set of mixed operations simulated on a production server. For both types of benchmarks we pick two of most popular ones to demonstrate HyCache performance: IOzone [81] and PostMark [89]. The third and fourth experiments are to test the functionality of HyCache with a real application. We achieve this by deploying MySQL and HDFS on top of HyCache, and execute TPC-H queries [189] on MySQL and the built-in 'sort' application of Hadoop, respectively.

Most experiments of HyCache+ are carried out on *Intrepid* [79], an IBM Blue Gene/P supercomputer of 160K cores at Argonne National Laboratory. We use up to 1024 nodes (4096 cores) in the evaluation. Each node has a 4-core PowerPC 450 processor (850 MHz) and 2 GB of RAM. A 7.6 PB GPFS [172] is deployed on 128 storage nodes. All experiments are repeated at least five times, or until results become stable (i.e. within 5% margin of error); the reported numbers are the average of all runs. Caching effect is carefully precluded by reading a file larger than the on-board memory before the measurement.

4.3.1 FUSE Overhead. To understand the overhead introduced by FUSE in HyCache, we compare the I/O performance between raw RAMDISK and a simple FUSE file system mounted on RAMDISK. By experimenting on RAMDISK we completely eliminate all factors affecting performance particularly from the spinning disk, disk controllers, etc. Since all the I/O tests are essentially done on the memory, any noticeable performance differences between the two setups are solely from FUSE itself.

We mount FUSE on */dev/shm*, which is a built-in RAMDISK in UNIX-like systems. The read and write bandwidth on both raw RAMDISK and FUSE-based virtual file system are reported in Figure 4.8. Moreover, the performance of concurrent FUSE processes are also plotted which shows that FUSE has a good performance scalability with respect to the number of concurrent jobs. In the case of single-process I/O, there is a significant performance gap between Ext4 and FUSE on RAMDISK. The read and write bandwidth of Ext4 on RAMDISK are in the order of gigabytes, whereas when mounting FUSE we could only get a bandwidth in the range of 500 MB/s. These results suggest that FUSE could not compete with the kernel-level file systems in raw bandwidth, primarily due to the overheads incurred by having the file system in user-space, the extra memory copies, and the additional context switching. However, we will see in the following subsections that even with FUSE overhead on SSD, HyCache still outperforms traditional spinning disks significantly, and that concurrency can be used to scale up FUSE performance close to the theoretical hardware performance (see Figure 4.11 and Figure 4.12).



(a) Read Bandwidth

(b) Write Bandwidth

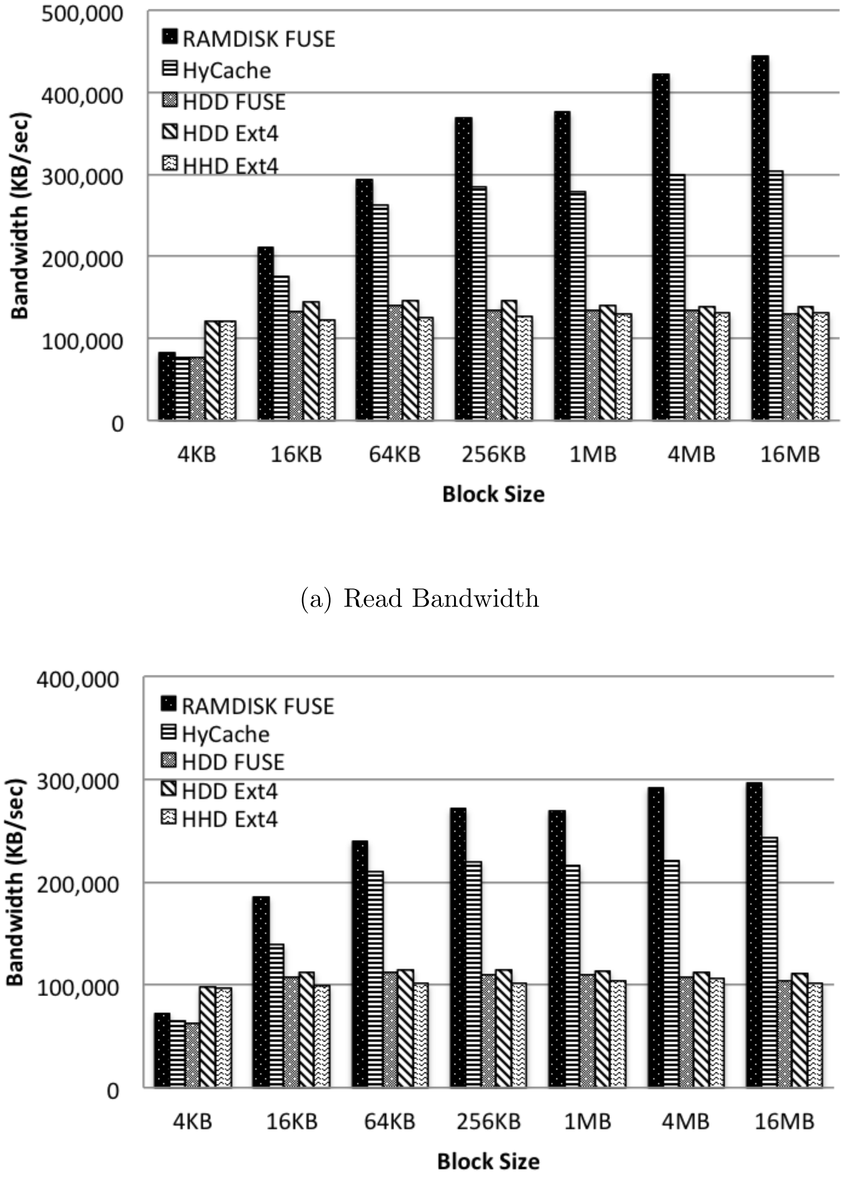
Figure 4.8. Bandwidth of raw RAMDISK and a FUSE file system mounted on RAMDISK. Px means x number of concurrent processes, e.g. FUSE RAMDISK P2 stands for 2 concurrent FUSE processes on RAMDISK.

4.3.2 HyCache Performance. IOzone is a general file system benchmark utility. It creates a temporary file with arbitrary size provided by the end user and then conducts a bunch of file operations like re-write, read, re-read, etc. In this chapter we use IOzone to test the read and write bandwidths as well as IOPS (input/output per second) on the different file systems.

We show the throughput with a variety of block sizes ranging from 4 KB to 16 MB. For each block size we show five bandwidths from the left to the right: 1) the theoretical bandwidth upper bound (obtained from RAMDISK), 2) HyCache, 3) a simple FUSE file system accessing a HDD, 4) HDD Ext4 and 5) HHD Ext4.

Figure 4.9(a) shows HyCache read speed is about doubled comparing to the native Ext4 file system for most block sizes. In particular when block size is 16 MB the peak read speed for HyCache is over 300 MB/s. It is 2.2X speedup with respect to the underlying Ext4 for HDD as shown in Figure 4.10(a). As for the overhead of FUSE framework compared to the native Ext4 file system on spinning disks we see FUSE only adds little overhead to read files at all block sizes as shown in Figure 4.10(a): for most block sizes FUSE achieves nearly 100% performance of the native Ext4. Similar results are also reported in a review of FUSE performance in [162]. This fact indicates that even when the SSD cache is intensive and some files need to be swapped between SSD and HDD, HyCache can still outperform Ext4 since the slower media of HyCache (HDD FUSE in Figure 4.9), are comparable to Ext4. We will present the application-level experimental results in the macro-benchmark subsection where we discuss the performance when files are frequently swapped between SSD and HDD. We can also see that the commercial HHD product performs at about the same level of the HDD, likely primarily due to a small and inexpensive SSD.

We see a similar result of file writes in Figure 4.9(b) as file reads. Again HyCache is about twice as fast when compared to Ext4 on spinning disks for most block sizes. The peak write bandwidth which is almost 250 MB/s is also obtained when block size is 16 MB, and it achieves 2.18x speedup for this block size compared



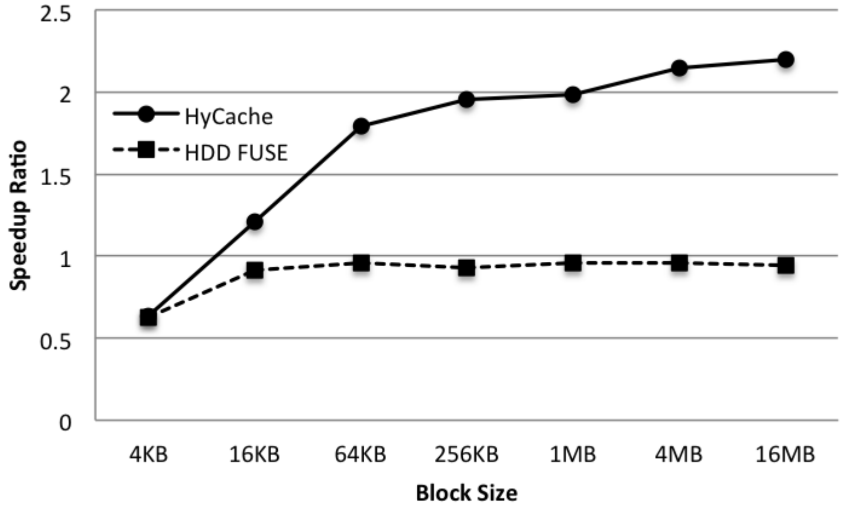
(a) Read Bandwidth

(b) Write Bandwidth

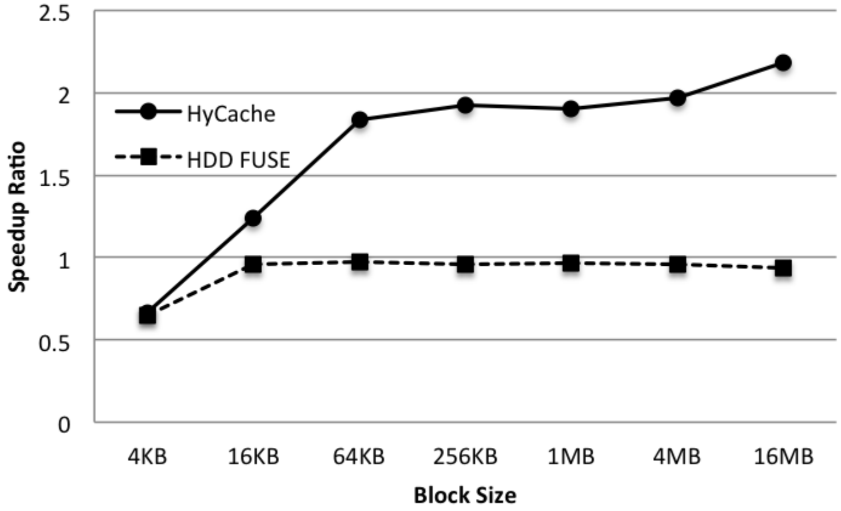
Figure 4.9. IOzone bandwidth of 5 file systems.

to Ext4 as shown in Figure 4.10(b). Also in this figure, just like the case of file reads we see little overhead of FUSE framework for the write operation on HDD except for 4KB block.

Figure 4.10 shows that for small block size (i.e. 4 KB) HyCache only achieves



(a) Read Speedup



(b) Write Speedup

Figure 4.10. HyCache and FUSE speedup over HDD Ext4.

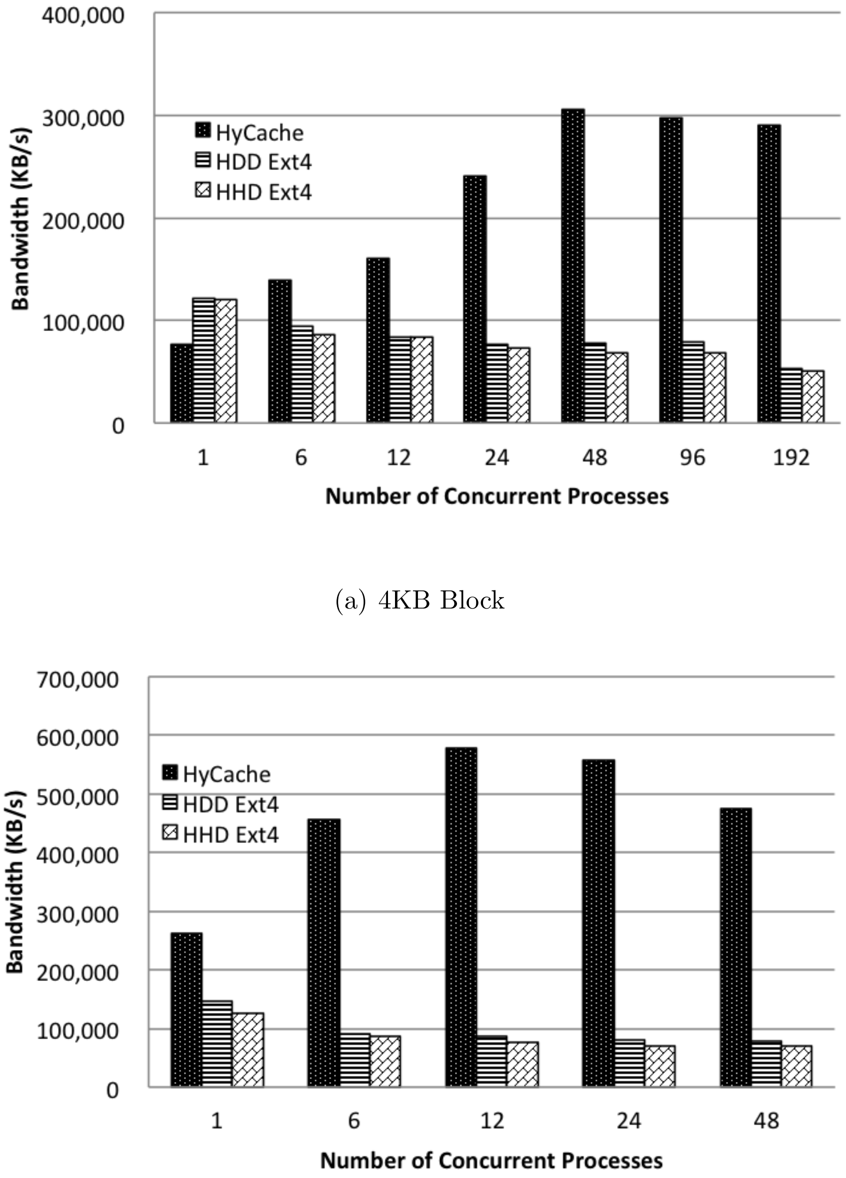
about 50% throughput of the native file system. This is due to the extra context switches of FUSE between user level and kernel level, in which the context switches of FUSE dominate the performance. Fortunately in most cases this small block size (i.e. 4 KB) is more generally used for randomly read/write of small pieces of data (i.e. IOPS) rather than high-throughput applications. Table 4.2 shows HyCache has a far higher IOPS than other Ext4. In particular, HyCache has about 76X IOPS as traditional HDD. The SSD portion of the HHD device (i.e. Seagate Momentus XT) is a read-only cache, which means the SSD cache does not take effect in this experiment because IOPS only involves random writes. This is why the IOPS of the HHD lands in the same level of HDD rather than SSD.

Table 4.2. IOPS of Different File Systems

HyCache	HDD Ext4	HHD Ext4
14,878	195	61

HyCache also takes advantages of the multicore's concurrent tasking which results in a much higher aggregate throughput. The point is that HyCache avoids reading/writing directly on the HDD so it handles multiple I/O requests concurrently. In contrast, traditional HDD only has a limited number of heads for read and write operations. Figure 4.11 shows that HyCache has almost linear scalability with the number of processes before hitting the physical limit (i.e. 306 MB/s for 4 KB block and 578 MB/s for 64 KB block) whereas the traditional Ext4 has degraded performance when handling concurrent I/O requests. The largest gap is when there are 12 concurrent processes for 64KB block (578 MB/s for HyCache and 86 MB/s for HDD): HyCache has 7X higher throughput than Ext4 on HDD.

The upper bound of aggregate throughput is limited by the SSD device rather than HyCache. This can be demonstrated in Figure 4.12 that shows how HyCache performs in RAMDISK. The performance of raw RAMDISK were also plotted. We can see that the bandwidth of 64KB block can be achieved at about 4 GB/s by concurrent processes. This indicates that FUSE itself is not a bottle neck in the last experiment: it will not limit the I/O speed unless the device is slow. This implies



(a) 4KB Block

(b) 64KB Block

Figure 4.11. Aggregate bandwidth of concurrent processes.

that HyCache can be applied to any faster storage devices in future as long as the workloads have enough concurrency to allow FUSE to harness multiple computing cores. Another observation is that HyCache can consume as much as 35% of raw memory bandwidth as shown in Figure 4.12 for 64KB block and 24 processes: 3.78 GB/s for HyCache and 10.80 GB/s for RAMDISK.

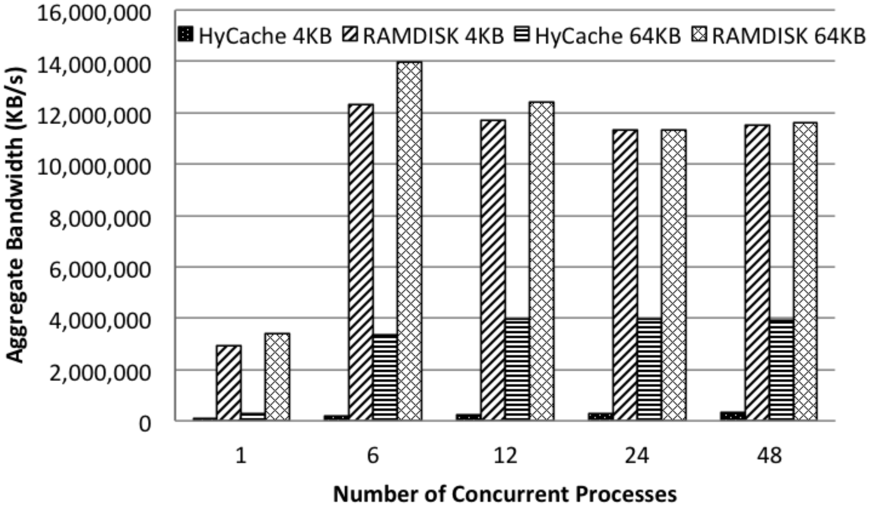
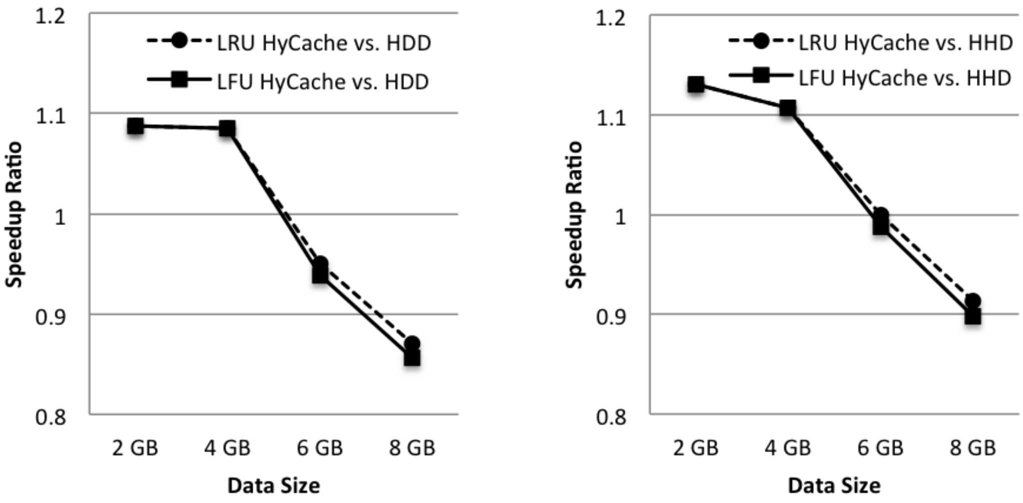


Figure 4.12. Aggregate bandwidth of the FUSE implementation on RAMDISK.

PostMark is one of the most popular benchmarks to simulate different workloads in file systems. It was originally developed to measure the performance of ephemeral small-file regime used by Internet software like Emails, netnews and web-based commerce, etc. A single PostMark instance carries out a number of file operations like read, write, append and delete, etc. In this chapter we use PostMark to simulate a synthetic application that performs different number of file I/Os on HyCache with two cache algorithms LRU and LFU, and compare their performances to Ext4.

We show PostMark results of four file systems: HyCache with LRU, HyCache with LFU, Ext4 on HDD and Ext4 on HHD. And for each of them we carried out four different workloads: 2 GB, 4 GB, 6 GB and 8 GB. To make a fair comparison between HyCache and the HHD device (i.e. Momentous XT: 4 GB SSD and 500 GB HDD), we set the SSD cache of HyCache to 4 GB. Figure 4.13 shows the speedup of HyCache with LRU and LFU compared to Ext4 on HDD and HHD. The difference between LRU and LFU is almost negligible (< 2%). The ratio starts to go down at 6

GB because HyCache only has 4 GB allocated SSD. Another reason is that PostMark only creates temporary files randomly without any repeated pattern. In other words it is a data stream making the SSD cache thrashes (this could be considered to be the worst case scenario).



(a) HyCache vs. HDD

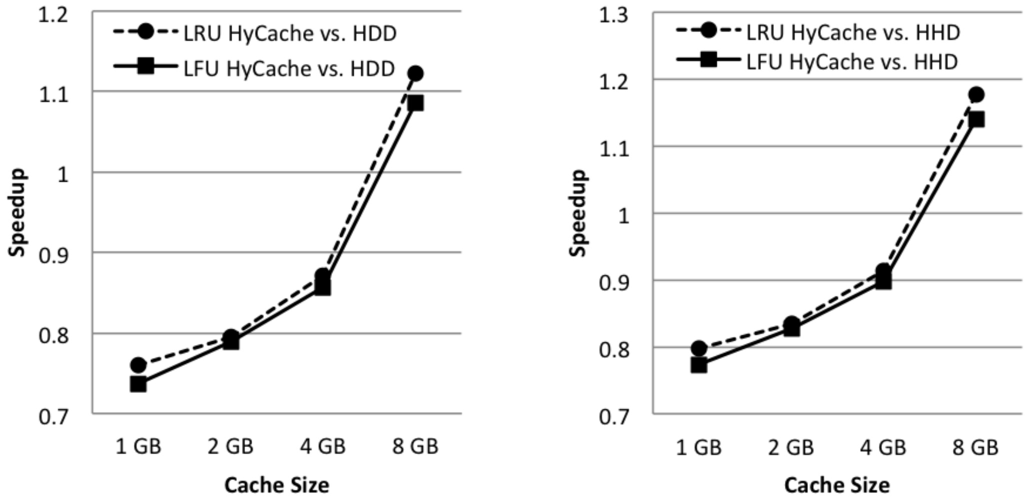
(b) HyCache vs. HHD

Figure 4.13. PostMark: speedup of HyCache over Ext4 with 4 GB SSD cache.

A big advantage of HyCache is that users can freely allocate the size of the SSD cache. In the last experiment HyCache did not work well as HHD mainly because the data is too large to fit in the 4 GB cache. Here we show how increasing the cache size impacts the performance. Figure 4.14 shows that if a larger SSD cache (i.e. 1GB - 8GB) is offered then the performance is indeed better than others with as much as a 18% performance improvement: LRU HyCache with 8GB SSD cache vs. HHD.

We have run two real world applications on HyCache: MySQL and the Hadoop.

We install MySQL 5.5.21 with database engine MySIAM, and deploy TPC-H 2.14.3 databases. TPC-H is an industry standard benchmark for databases. By default it provides a variety size of databases (e.g. scale 1 for 1 GB, scale 10 for

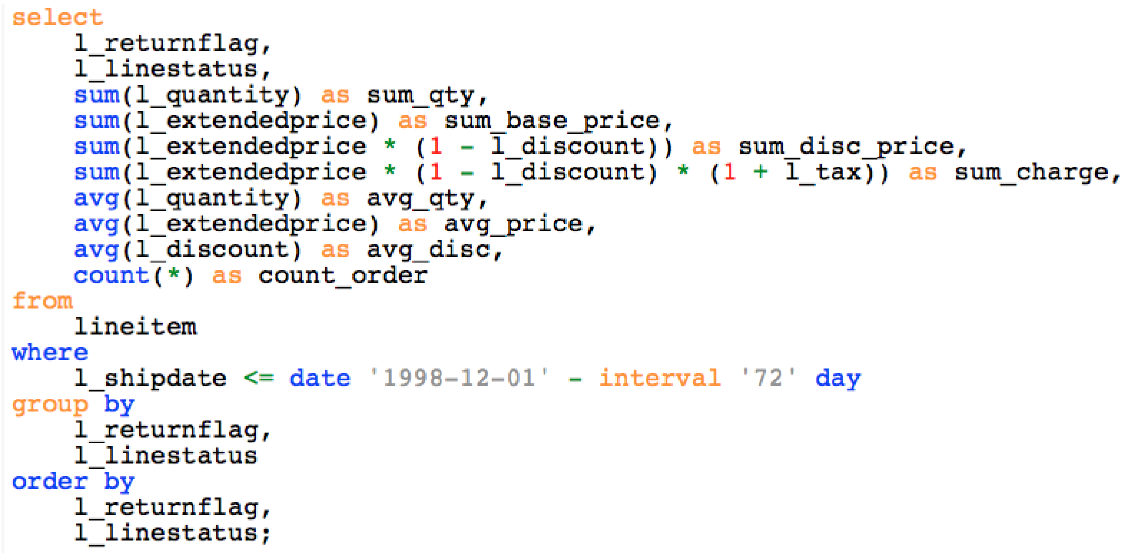


(a) HyCache vs. HDD

(b) HyCache vs. HHD

Figure 4.14. PostMark: speedup of HyCache with varying sizes of cache.

10 GB, scale 100 for 100GB) each of which has eight tables. Furthermore, TPC-H provides 22 complicated queries (i.e. Query #1 to Query #22) that are comparable to business applications in the real world. Figure 4.15 shows Query #1 which will be used in our experiments.



```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
    l_shipdate <= date '1998-12-01' - interval '72' day
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
```

Figure 4.15. TPC-H: Query #1.

To test file writes in HyCache, we loaded table *lineitem* at scale 1 (which is about 600 MB) and scale 100 (which is about 6 GB) in these three file systems: LRU HyCache, HDD Ext4 and HHD Ext4. As for file reads we ran Query #1 at scale 1 and scale 100. HyCache has an overall of 9% and 4% improvement over Ext4 on HDD and HHD, respectively. The result details of these experiments are reported in Figure 4.16.

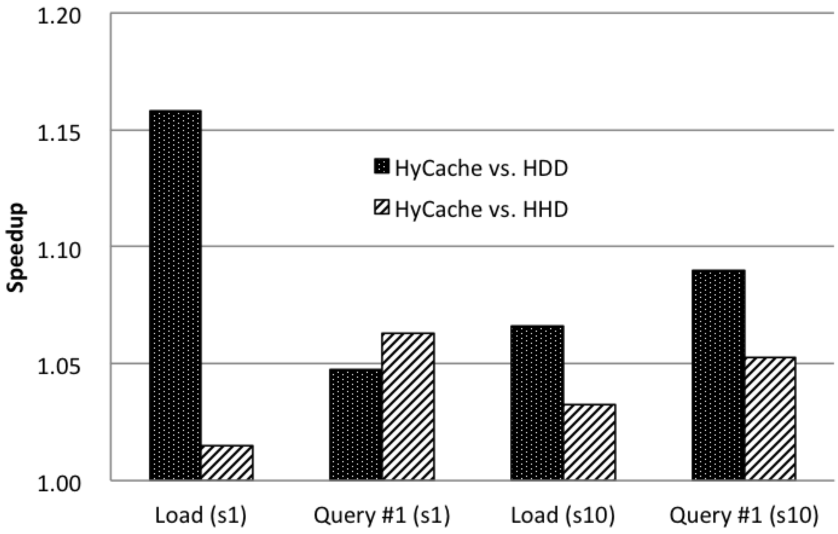


Figure 4.16. TPC-H: speedup of HyCache over Ext4 on MySQL.

For HDFS we measure the bandwidth by concurrently copying a 1GB file per node from HDFS to the RAMDISK (i.e. */dev/shm*). The results are reported in Table 4.3, showing that HyCache helps improve HDFS performance by 28% at 32-node scales.

We also run the built-in 'sort' example as a real Hadoop application. The 'sort' application is to use map-reduce [43] to sort a 10GB file. We kept all the default settings in the Hadoop package except for the temporary directory which is specified as the HyCache mount point or a local Ext4 directory. The results are reported in Table 4.3.

Table 4.3. HDFS Performance

	w/o HyCache	w/ HyCache	Improvement
bandwidth	114 MB/sec	146 MB/sec	28%
sort	2087 sec	1729 sec	16%

4.3.3 HyCache+ Performance. We illustrate how HyCache+ significantly improves the I/O throughput of parallel file systems. The local cache size is set to 256 MB (0.25 GB) on each node. To measure local cache's stable throughput, each client repeatedly writes a 256 MB file for 63 times (total 15.75 GB). Then another 256 MB file is written on each client to trigger the swapping between local cache and GPFS. Figure 4.17 reports (at 256-core scale) the real-time aggregate throughput, showing a significant performance drop at around 90-second timestamp, when the 15.75GB data are finished on the local cache.

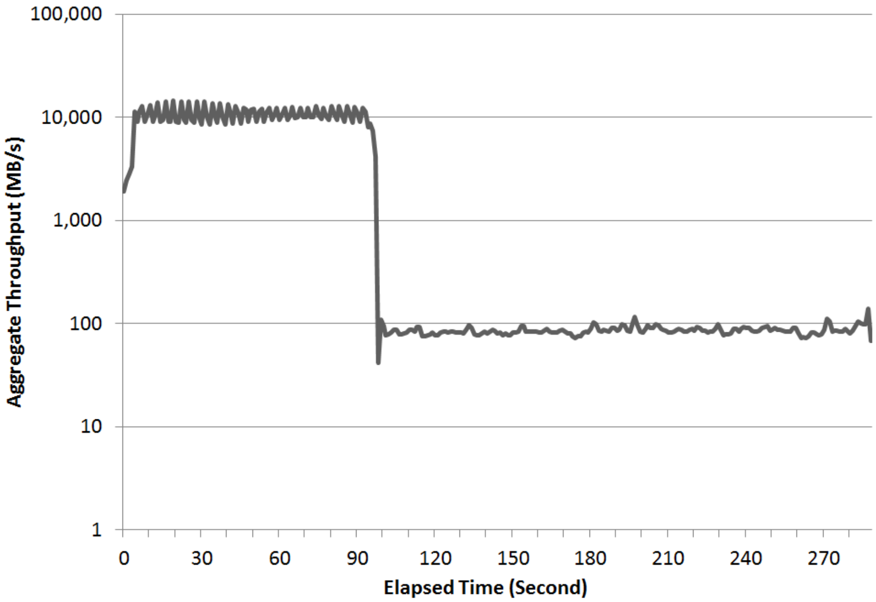


Figure 4.17. Throughput on Blue Gene/P (256 cores)

We demonstrate the scalability of HyCache+ by repeating the experiment of the same per-client workload on 512 nodes (2048 cores). The real-time throughput is reported in Figure 4.18. We see that HyCache+ shows an excellent scalability for both the cached data and the remote data: both the caching throughput and the disk throughput are about 8X faster than those numbers at 256-core scale in Figure 4.17.

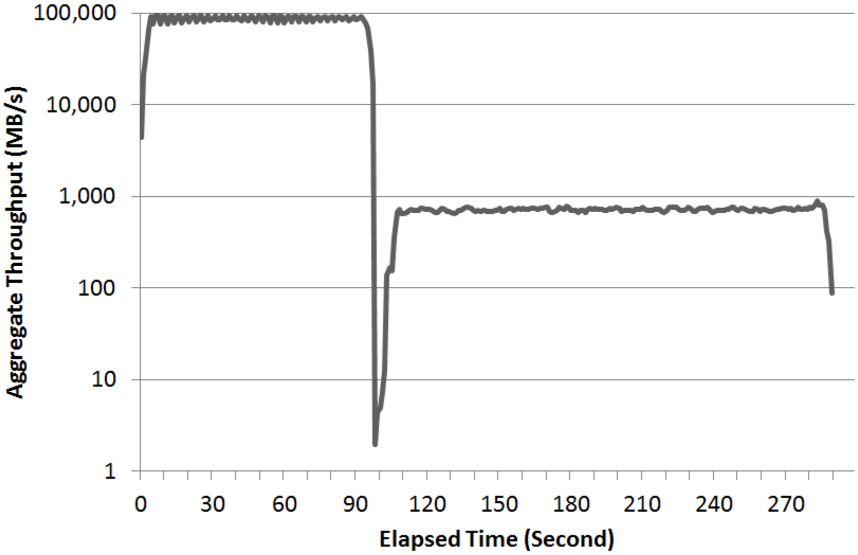


Figure 4.18. Throughput on Blue Gene/P (2048-cores)

We plug the heuristic caching and LRU algorithms into HyCache+, and simulate their performance at 512-node scale on Intrepid. We create different sizes of data, randomly between 6MB and 250MB, and repeatedly read these data in a round-robin manner. The local cache size is set to 256MB. The execution time of both algorithms is reported in Figure 4.19. Heuristic caching clearly outperforms LRU at all scales, mainly because LRU does not consider the factors such as file size and cost-gain ratio, which are carefully taken into account in heuristic caching. In particular, heuristic caching outperforms LRU by 29X speedup at I/O size = 64,000GB (3,009 seconds vs. 86,232 seconds).

4.4 Summary

This chapter presents HyCache that addresses the long-existing issue with the

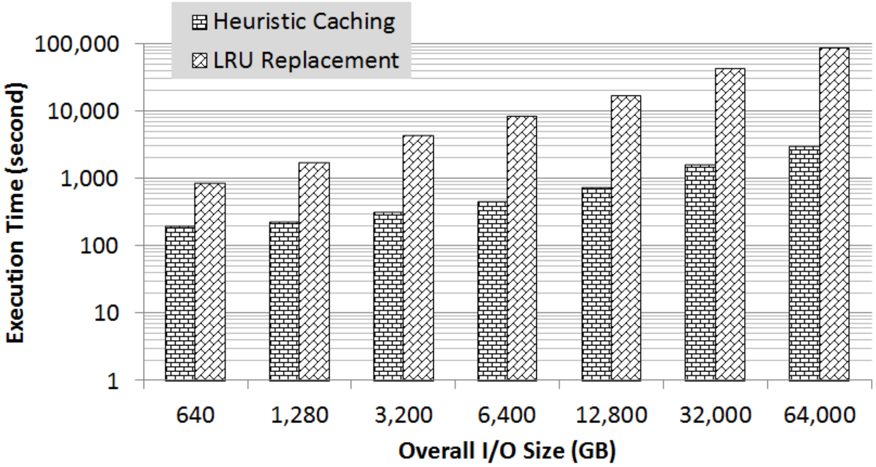


Figure 4.19. Comparison between Heuristic Caching and LRU

bottleneck of local spinning hard drives in distributed file systems and proposed a cost-effective solution to alleviate this bottleneck, aimed at delivering comparable performance of an all SSD solution at a fraction of the cost. We proposed to add a middleware layer between the distributed filesystem and the underlying local file systems. We designed and implemented HyCache with high throughput, low latency, strong consistency, single namespace, and multithread support. Non-privileged users can specify the cache size for different workloads without modifying the applications or the kernel. Our extensive performance evaluation showed that HyCache can be competitive with kernel-level file systems, and significantly improves the performance of the upper-level distributed file systems.

We then extend HyCache to HyCache+, a scalable high-performance caching middleware to improve the I/O performance of parallel filesystems. A novel 2-layer approach is proposed to minimize the network cost and heuristically optimize the caching effect. Large scale evaluation at up to 4096 cores shows that HyCache+ improves the I/O performance by up to two orders of magnitude, and the proposed caching approach could further elevate the performance by 29X.

CHAPTER 5
LIGHTWEIGHT PROVENANCE

It has become increasingly important to capture and understand the origins and derivation of data (its provenance). A key issue in evaluating the feasibility of data provenance is its performance, overheads, and scalability.

In this chapter, we explore the feasibility of a general metadata storage and management layer for parallel file systems, in which metadata includes both file operations and provenance metadata. We experimentally investigate the design optimality: whether provenance metadata should be loosely-coupled or tightly integrated with a file metadata storage systems. We consider two systems that have applied similar distributed concepts to metadata management, but focusing singularly on kind of metadata: (i) FusionFS, which implements a distributed file metadata management based on distributed hash tables, and (ii) SPADE, which uses a graph database to store audited provenance data and provides distributed module for querying provenance. Results were published in [177, 227].

5.1 Background

Scientific advancement and discovery critically depends upon being able to extract knowledge from extremely large data sets, produced either experimentally or computationally. In experimental fields such as high-energy physics datasets are expected to grow by six orders of magnitude [58]. To extract knowledge from extremely large datasets in a scalable way, architectural changes to HPC systems are increasingly being proposed—changes that either reduce simulation output data [112, 111] or optimize the current flop to I/O imbalance [63, 5].

A primary architectural change is a change in the design of the storage layer, which is currently segregated from compute resources. Storage is increasingly being placed close to compute nodes in order to help manage large-scale I/O volume and data movement, especially for efficient checkpointing at extreme scale. This change in the storage layer has a significant resulting advantage—it enables simulation output data to be stored with the provenance metadata so that analysis can be easily verified, validated as well as retraced over time steps even after the simulation has finished.

While this architectural change is being deemed necessary to provide the much needed scalability advantage of concurrency and throughput, it cannot be achieved without providing an efficient storage layer for conducting metadata operations. The centralized metadata repository in parallel file systems has shown to be inefficient at large scale for conducting metadata operations, growing for instance from tens of milliseconds on a single node (four-cores), to tens of seconds at 16K-core scales [159, 216]. Similarly, auditing and querying of provenance metadata in a centralized fashion has shown poor performance over distributed architectures [114].

In this chapter, we explore the feasibility of a general metadata storage and management layer for parallel file systems, in which metadata includes both file operations and provenance metadata. In particular we experimentally investigate the design optimality—whether provenance metadata should be loosely-coupled or tightly integrated with a file metadata storage systems. To conduct this experimental evaluation, we consider two systems that have applied similar distributed concepts to metadata management, but focusing singularly on kind of metadata: (i) FusionFS [225], which implements a distributed file metadata management based on distributed hash tables, and (ii) SPADE [61], which uses a graph database to store audited provenance data and provides distributed module for querying provenance.

Both FusionFS and SPADE are good choices for investigating the metadata storage design problem since both systems have similar manifestation of distributed concepts towards storing their individual metadata: (1) FusionFS provides a POSIX interface which makes a perfect corresponding for SPADE user-level file system (FUSE-based) provenance collection; (2) both systems work in a decentralized way thus actively exploiting the resources at each node.

The remainder of this chapter first introduces the `SPADE+FusionFS` version of provenance-aware distributed file system, that aims to offer excellent scalability while retaining the provenance overhead negligible in traditional clusters. Some preliminary results of `SPADE+FusionFS` have been published in [177]. We then investigate Zero-hop Distributed Hashtable (ZHT) [101] as the underlying storage system for provenance [227]. ZHT is currently used to store file metadata in FusionFS and provides the following features that makes it a desirable choice to store provenance: (1) excellent storage load balancing; (2) light-weighted and fast; (3) excellent scalability; (4) be able to provide a global view of provenance that aims to provide provenance capture and management in petascale and exascale. We term the ZHT-backed provenance system as `FusionProv`.

5.2 Local Provenance Middleware

5.2.1 SPADE. SPADE is a software infrastructure for data provenance collection, management, and analysis. Different operating system level *reporters* facilitate provenance collection. The underlying data model is graph-based, consisting of vertices and directed edges, each of which can be labeled with an arbitrary number of annotations (in the form of key-value pairs). These annotations can be used to embed the domain-specific semantics of the provenance. The SPADE system decouples the production, storage, and utilization of provenance metadata, as illustrated in Figure 5.1. At its core is a provenance kernel that mediates between the producers and consumers of provenance information, and handles the persistent storage of records. The kernel handles buffering, filtering, and multiplexing incoming metadata from multiple provenance sources. It can be configured to commit the elements to multiple databases, and responds to concurrent queries from local and remote clients.

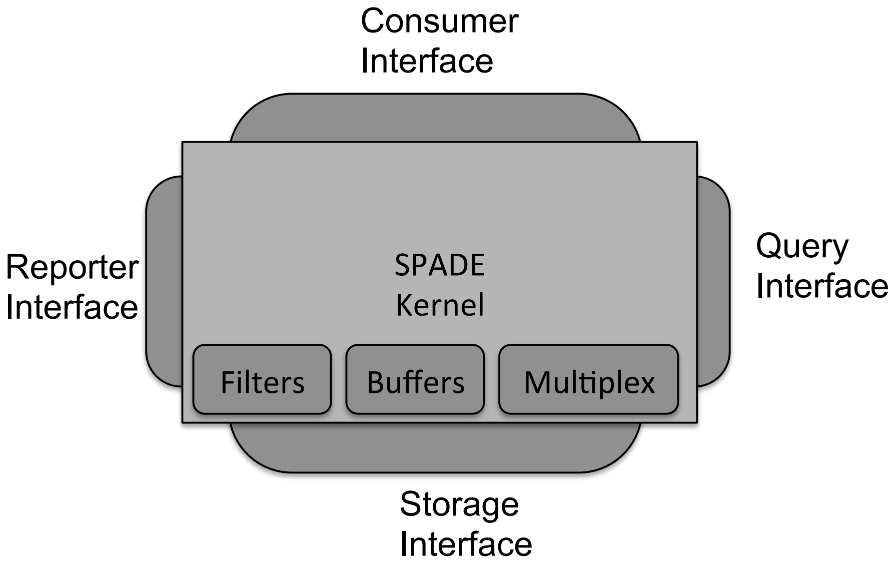


Figure 5.1. The SPADE architecture

5.2.2 Design. The architecture of `SPADE+FusionFS` integration is shown in Figure 5.2. Each node has two services installed: FusionFS service and SPADE service. One service type can only communicate to the other type on the local node. That is, a SPADE service only communicates with its local FusionFS service, and vice versa. For services of the same type (e.g. FusionFS ⇔ FusionFS, SPADE ⇔ SPADE), they are free to talk to others remotely.

In order to make the collected provenance compliant to the Open Provenance Model (OPM), when there is a network transmission, SPADE creates a "dummy" FusionFS process vertex to connect two artifacts: a file vertex and a network vertex. We call it a "dummy" process because clients do not need to be concerned with this process when querying provenance; it is just a symbol to indicate the network transmission is triggered by FusionFS in OPM. Figure 5.3 shows how a network

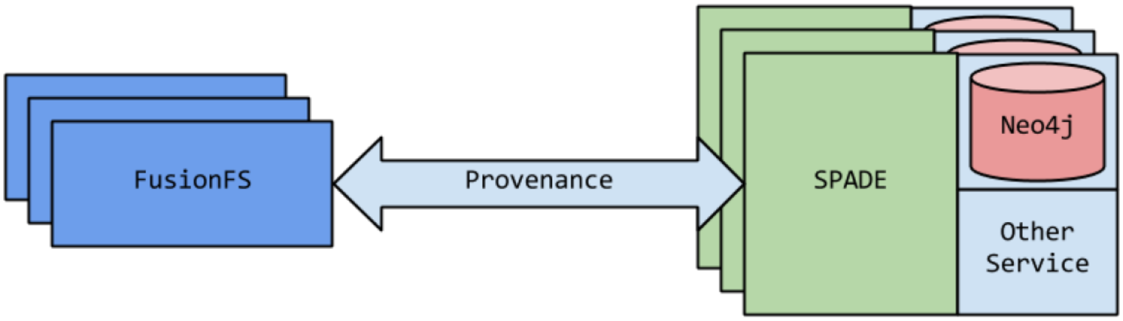


Figure 5.2. FusionFS+SPADE architecture overview

transmission is represented.

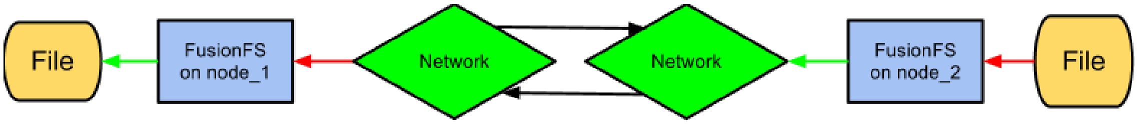


Figure 5.3. Network Transmission

5.2.3 Implementation. The key challenge of the proposed work is how to seamlessly integrate SPADE and FusionFS. All communication between these two services is implemented with TCP. Asynchronous communication is not used because of the short life cycle of some processes. SPADE collects parts of the process information based on system files under directory /proc/pid. If a process starts and terminates too fast for SPADE to catch, there would be provenance loss. Therefore it is critical to keep synchronous communication between SPADE and FusionFS, at least while the two systems are completely decoupled. We hope to address this in future work with a tighter integration between FusionFS and SPADE.

Most communication between SPADE and FusionFS consists of simple operation bindings. For example, FusionFS write operation invokes SPADE to collect write provenance for this operation. However, as a distributed file system, FusionFS sometimes needs to migrate files between nodes. The original network provenance collection in SPADE is not optimized for FusionFS. So we make some customization to the network provenance collection to fully hide unnecessary provenance data outside FusionFS.

5.2.4 Provenance Granularity. One common practice in file manipulations is to split (large) files into blocks to improve the space efficiency and responsive time. However, for the purpose of provenance, it is less interesting to keep track of file traces at the block level: in most cases, a file-level provenance would suffice. We have implemented both the file-level and the block-level provenance tracings, namely, the fine-grained provenance and the coarse-grained provenance.

5.3 Distributed Provenance

5.3.1 Design. Figure 5.4 illustrates how we integrate FusoinFS and ZHT to support distributed provenance capture at the file system level. Provenance is firstly generated in the FUSE layer in FusionFS, and then is cached in the local provenance buffer. And at a certain point (e.g. when the file is closed), the cached provenance will be persisted into ZHT. Users can do query on any node of the system using a ZHT client.

5.3.2 Implementation. Table 5.1 shows what is captured for the graph vertex in the distributed provenance store. Basically there are two different vertex types being tracked of: file and process. In other words, we are interested in which file(s) have been touched by which process(es). And we maintain a linked list for the tree topology in ZHT.

We provide a set of APIs to allow users plug their own implementations for the provenance they are interested in. Some commonly used APIs are listed in Table 5.2. Note that for file creation, there is no need to save the provenance in the local buffers

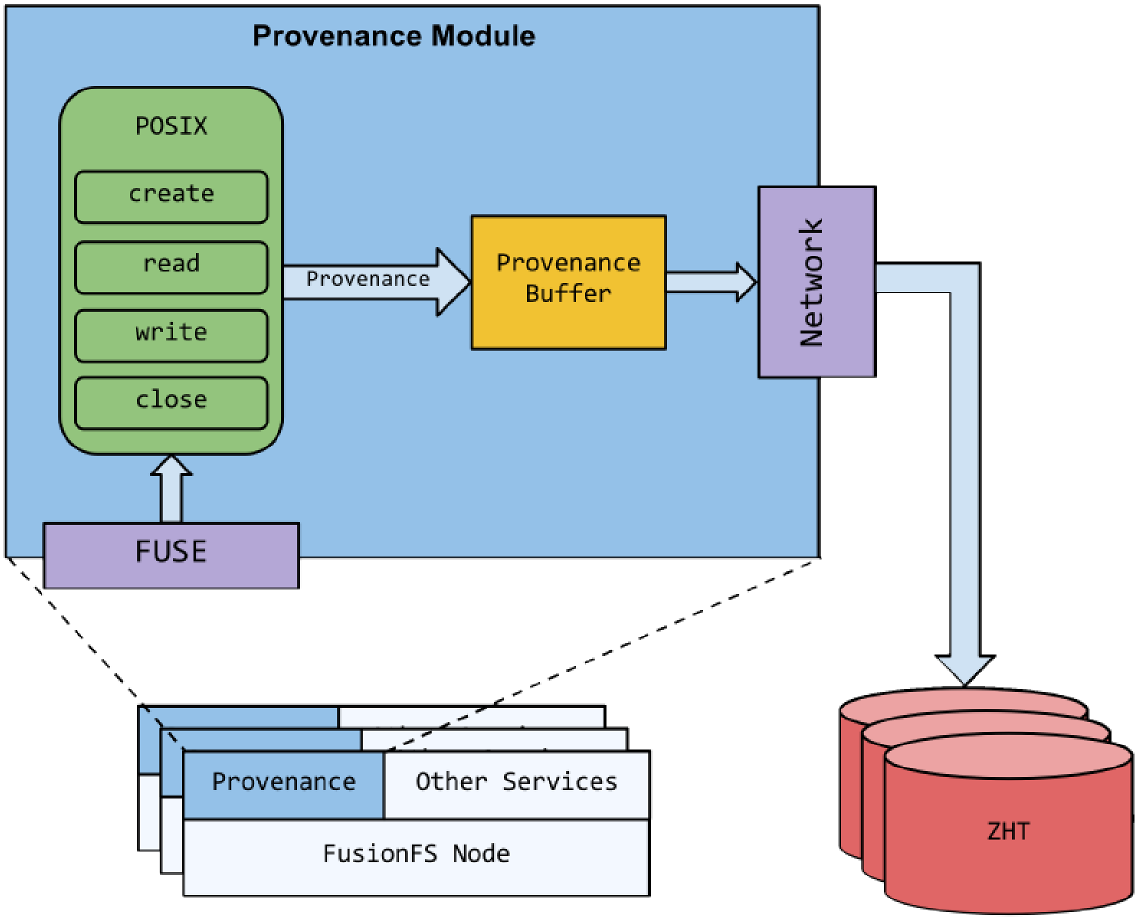


Figure 5.4. FusionFS+ZHT architecture overview

Table 5.1. Attributes of Graph Vertexes in Distributed Provenance Capture

Vertex Type	Attributes
File	[File path/name] [File version] [File size]
Process	[Process host] [Process name] [Process command line]

because it only touches the metadata (rather than the file/process). Therefore this information is directly stored in the underlying metadata storage (i.e. ZHT).

We implement a light-weight command-line tool that end users can use to

Table 5.2. Some Example APIs Available for Provenance Capture

FusionFS Operation	Provenance API
fusion_read()	prov_read()
fusion_write()	prov_write()
fusion_create()	prov_create()
fusion_flush()	prov_push()

query the provenance, in the following syntax:

```
query vertex [file] [version] [ancestors--descendants] [depth]
```

For example, with a following workflow: a file (origin_file) was created by a touch process on host 12.0.0.1, and later was copied by multiple processes on multiple nodes (12.0.0.2 to 12.4.0.16). The query on the descendants of the touch process (vertex) would generate provenance graph showed in Figure 5.5.

5.4 Experiment Results

We have deployed the distributed provenance-aware file system on 1K-node IBM BlueGene/P supercomputer Intrepid [79]. We also evaluated both the distributed and SPADE-extended systems on a 32-node cluster, where each node has two Quad-Core AMD Opteron 2.3GHz processors with 8GB memory. All nodes are interconnected by 1Gbps Ethernet. All experiments are repeated at least 3 times to obtain stable results (i.e. within 5% difference).

5.4.1 SPADE + FusionFS.

5.4.1.1 Single-Node Throughput. We first measured the performance of provenance collection within FusionFS on a single node. A client reads/writes a 100MB

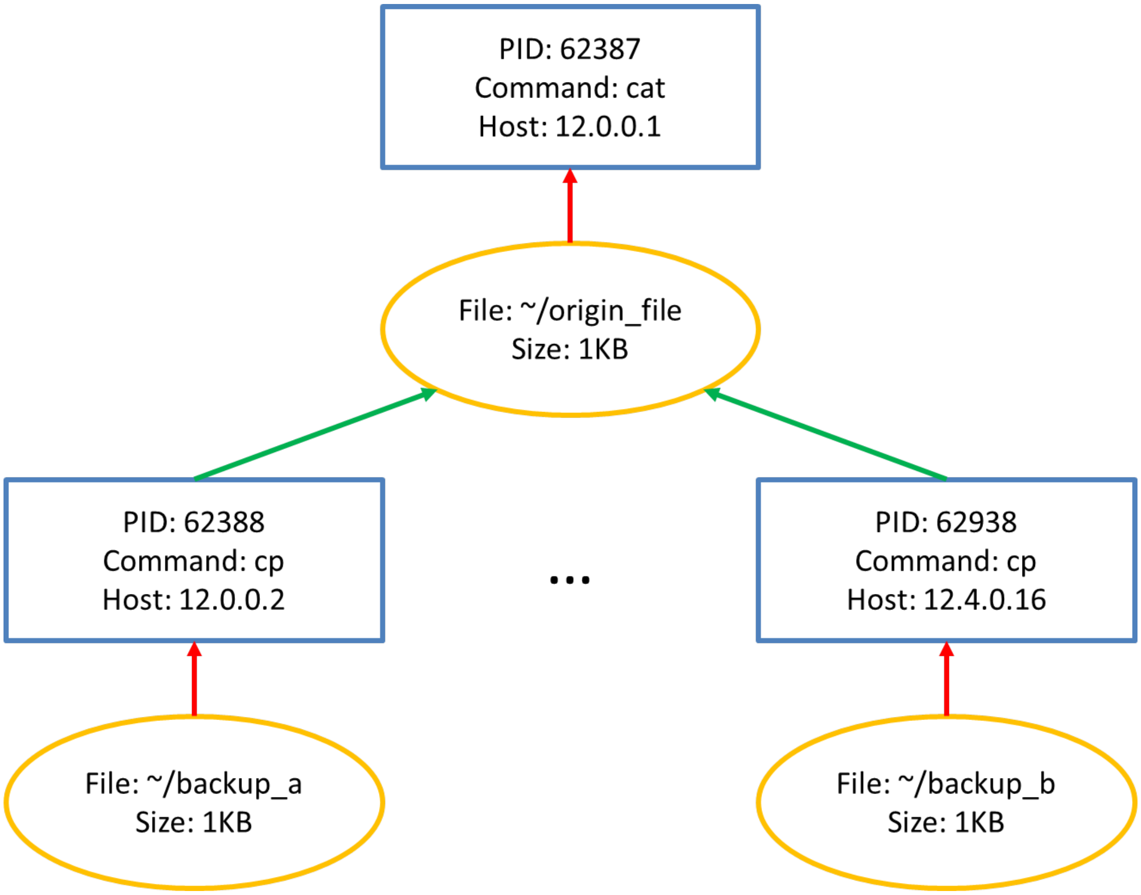


Figure 5.5. An example query tree in distributed provenance systems

file from/to FusionFS. We compare the performance between fine-grained and coarse-grained provenance collection with different block sizes. The benchmark we used is IOZone [81], which is carefully tuned to avoid operating system cache.

Figure 5.6 and Figure 5.7 show that a fine-grained provenance collection introduces a high overhead. Even though a larger block size could reduce the overhead to some degree, the number is still significantly high (i.e. around 75%), compared to coarse-grained provenance (i.e. less than 5%). This is expected since a bigger I/O block size results in fewer I/O runs, which further involves less time to collect provenance (SPADE spends on average 2.5 ms for each provenance recording, which corresponds to a single I/O run in the fine-grained provenance collection).

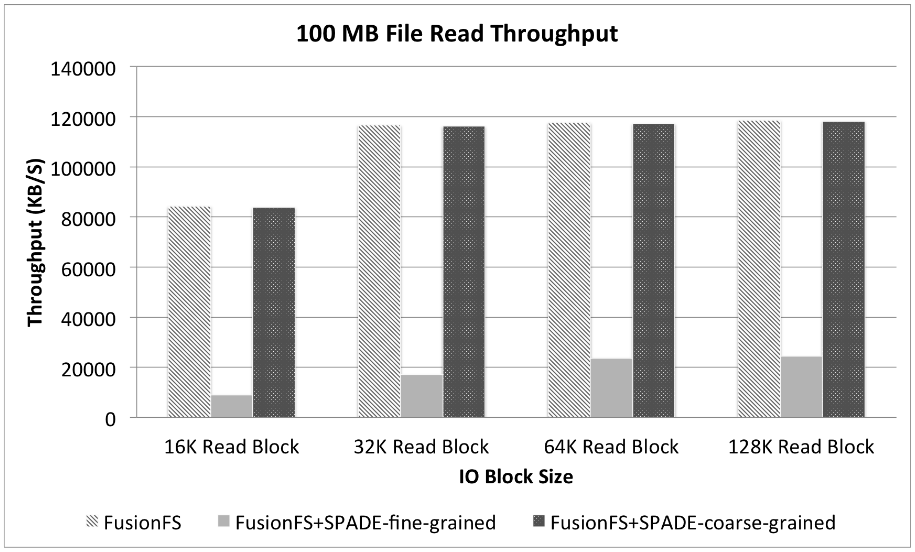


Figure 5.6. Read Throughput

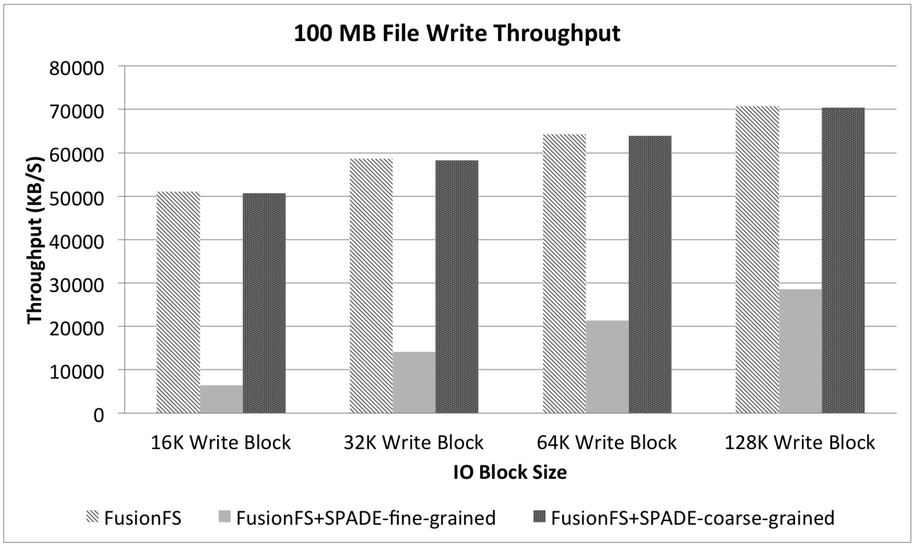


Figure 5.7. Write Throughput

5.4.1.2 Multi-Node Throughput. In the 32-node cluster, multiple clients read/write distinct files from/to FusionFS. The file size is set to 100MB and the I/O block size is set to 128KB.

In Figure 5.8, a coarse-grained provenance collection shows a much better performance than the fine-grained counterpart (consistent with the single-node benchmark results). Both fine-grained and coarse-grained provenance show excellent scalability with linear increase in performance. This can be explained by two facts: (1) SPADE only collects provenance of the local node, and (2) FusionFS scales linearly with respect to the number of nodes by getting high data locality in the data access pattern evaluated. We have evaluated FusionFS (without SPADE) at scales of up to 1K nodes on a IBM Blue Gene/P supercomputer with similar excellent results.

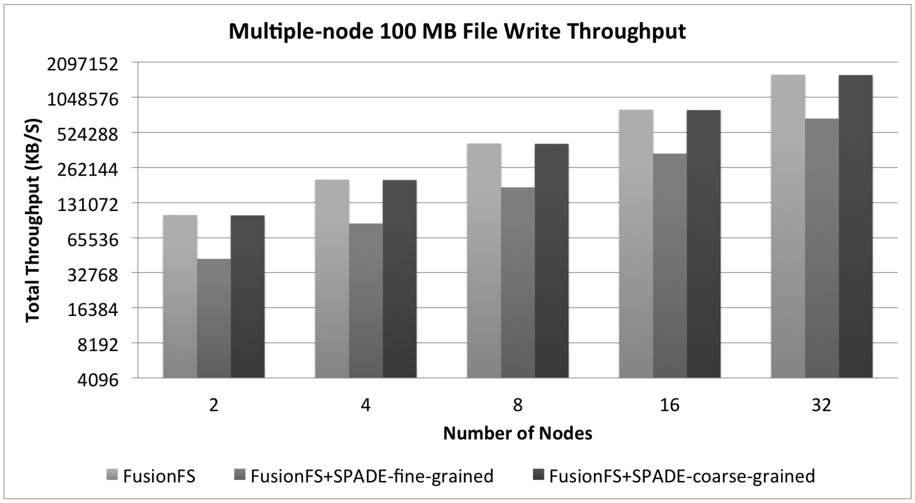


Figure 5.8. Multiple-Node 100MB Write Throughtput

5.4.1.3 Query time. We are interested in the query time of the provenance of a particular file that has been read by multiple remote nodes. This write-once-read-many is a very frequent pattern in the context of a distributed system. The query is shown in the following format:

$$\texttt{query lineage descendants } vertex - id \texttt{ 100 null filename:} test.file.name$$

Since SPADE (with version) does not support executing sub-query in parallel, the total query time increases as it scales up. However, according to Figure 5.9, with different scales from 2 to 32 nodes, the average per-node query time is about constant, indicating that adding more nodes will not put more burden to the provenance system. This is expected, since the underlying FusionFS has an excellent scalability and SPADE on each node adds negligible overheads locally.

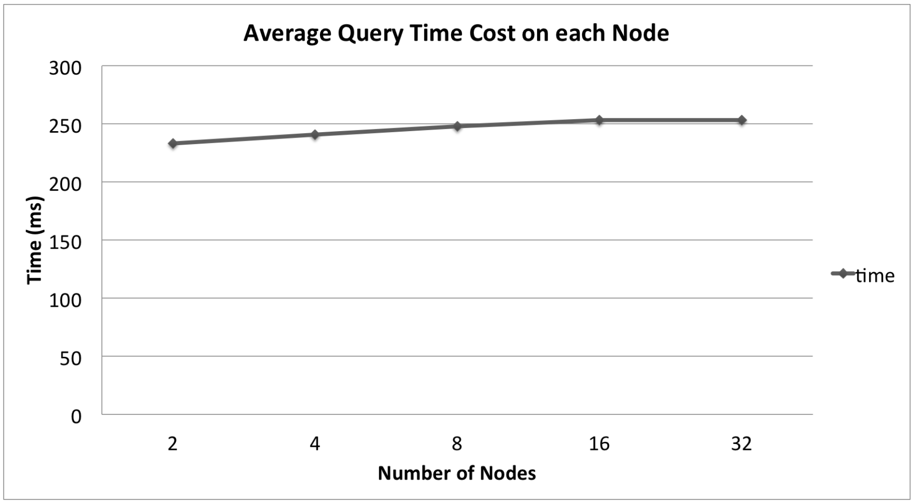


Figure 5.9. Query Time Cost

5.4.2 Distributed Provenance Capture and Query.

5.4.2.1 Provenance capture. We compare the throughput of the distributed provenance capture to the `SPADE+FusionFS` implementation in Figure 5.10. The ZHT-based throughput is comparable to both the pure FusionFS and the coarse-grained `SPADE+FusionFS` implementations. This result suggests that, even though there is network overhead involved in distributed provenance capture, the cost is about negligible.

5.4.2.2 Provenance Query. Similarly to throughput, we also compare the query time of different implementations. Figure 5.11 shows that even on one single node, the ZHT-based implementation is much faster than SPADE (0.35ms vs. 5ms). At 32-node scale, the gap is even larger result in 100X difference (108ms vs. 11625ms).

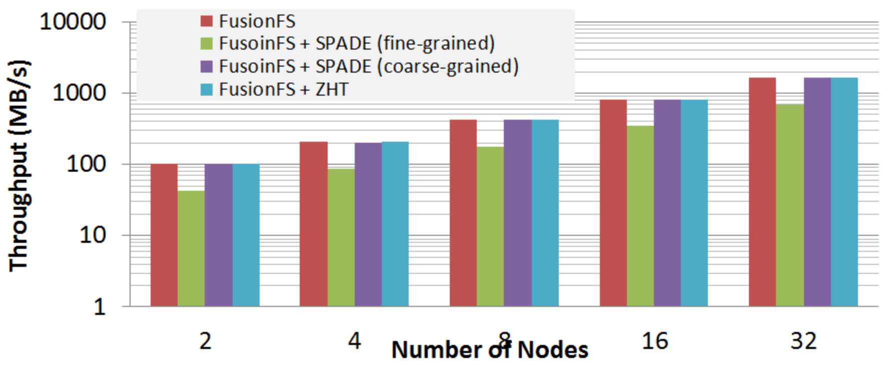


Figure 5.10. Throughput of different implementations of provenance capture

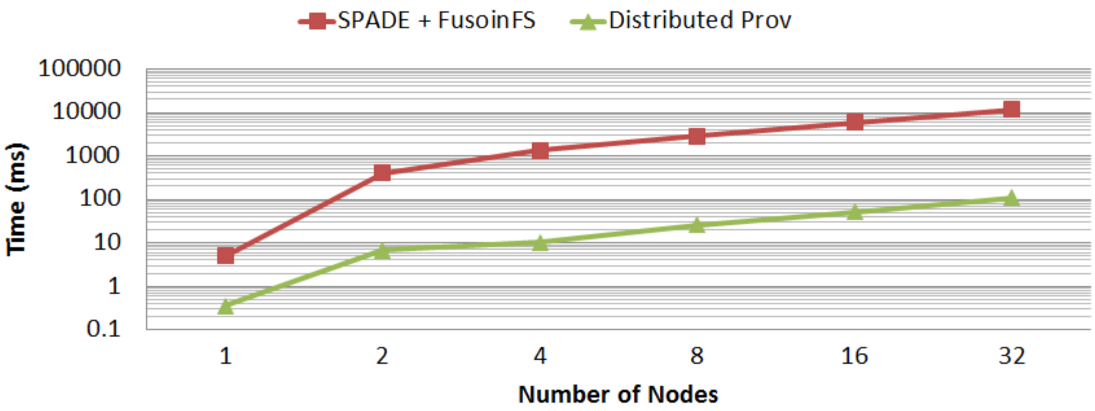


Figure 5.11. Query time of different provenance system

5.4.2.3 Scalability. We have scaled the distributed provenance system up to 1K-node on IBM Blue Gene/P. Figure 5.12 shows that the provenance overhead is relative small even on 1K nodes (14%). Similarly, we report the query time and overhead on the same workload at large scale (i.e. 1K nodes) in Figure 5.13, which shows that the overhead at 1K-nodes is about 18%.

5.5 Summary

This chapter explores the feasibility of a general metadata storage and management layer for parallel file systems, in which metadata includes both file operations

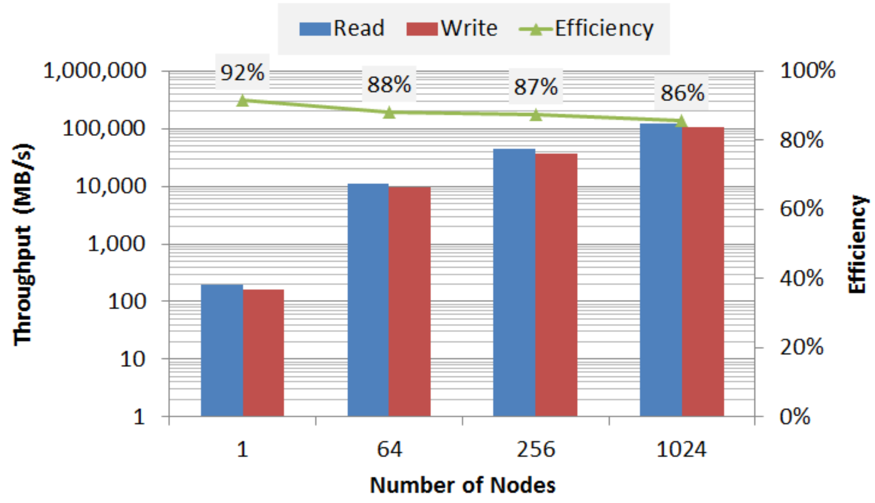


Figure 5.12. Throughput on Blue Gene/P

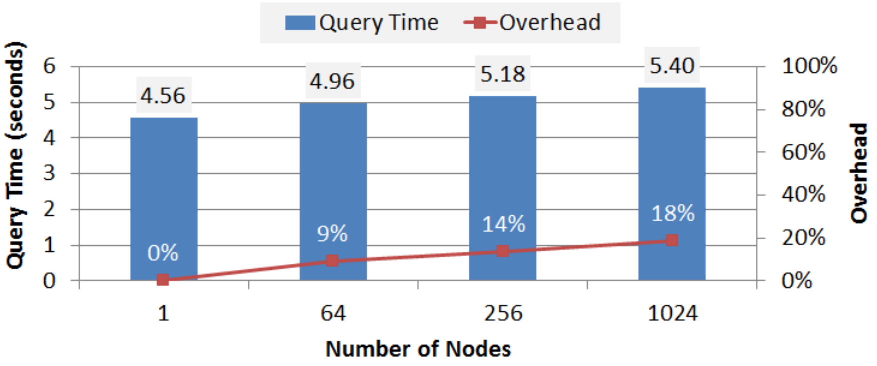


Figure 5.13. Query Time on Blue Gene/P

and provenance metadata. Two systems are investigated (1) FusionFS, which implements a distributed file metadata management based on distributed hash tables, and (2) SPADE, which uses a graph database to store audited provenance data and provides distributed module for querying provenance. Our results on a 32-node cluster show that FusionFS+SPADE is a promising prototype with negligible provenance overhead and has promise to scale to petascale and beyond. Furthermore, FusionFS with its own storage layer for provenance capture is able to scale up to 1K nodes on

Blue Gene/P supercomputer.

CHAPTER 6
TRANSPARENT COMPRESSION

Data compression could ameliorate the I/O pressure of scientific applications on high-performance computing systems. Unfortunately, the conventional wisdom of naively applying data compression to the file or block brings the dilemma between efficient random accesses and high compression ratios. File-level compression can barely support efficient random accesses to the compressed data: any retrieval request need trigger the decompression from the beginning of the compressed file. Block-level compression provides flexible random accesses to the compressed data, but introduces extra overhead when applying the compressor to each every block that results in a degraded overall compression ratio.

This chapter introduces a concept called *virtual chunks* [231, 232, 224] aiming to support efficient random accesses to the compressed scientific data without sacrificing its compression ratio. In essence, virtual chunks are logical blocks identified by appended references without breaking the physical continuity of the file content. These additional references allow the decompression to start from an arbitrary position (efficient random access), and retain the file's physical entirety to achieve high compression ratio on par with file-level compression.

6.1 Background

As today's scientific applications are becoming data-intensive, one effective approach to relieve the I/O bottleneck of the underlying storage system is data compression. As a case in point, it is optional to apply lossless compressors (e.g. LZO [110], bzip2 [24]) to the input or output files in the Hadoop file system (HDFS) [178], or even lossy compressors [96, 95] at the high-level I/O middleware such as HDF5 [73] and NetCDF [136]. By investing some computational time on compression, we hope to significantly reduce the file size and consequently the I/O time to offset the computational cost.

State-of-the-art compression mechanisms of parallel and distributed file systems, however, simply apply the compressor to the data either at the file-level or block-level[1], and leave the important factors (e.g. computational overhead, compression ratio, I/O pattern) to the underlying compression algorithms. In particular, we observe the following limitations of applying the file-level and block-level compression, respectively:

1. The file-level compression is criticized by the significant overhead for random accesses: the decompression needs to start from the very beginning of the compressed file anyway even though the client might be only requesting some bytes at an arbitrary position of the file. As a case in point, one of the most commonly used operations in climate research is to retrieve the latest temperature of a particular location. The compressed data set is typically in terms of hundreds of gigabytes; nevertheless scientists would need to decompress the entire compressed file to only access the last temperature reading. This wastes both the scientist's valuable time and scarce computing resources.

2. The deficiency of block-level compression stems from its additional compression overhead larger than the file-level counterpart, resulting in a degenerated compression ratio. To see this, think about a simple scenario that a 64MB file to be compressed with 4:1 ratio and 4 KB overhead (e.g. header, metadata, etc.). So the resultant compressed file (i.e. file-level compression) is about 16MB+4KB = 16.004MB. If the file is split into 64KB-blocks each of which is applied with

[1] The "chunk", e.g. in HDFS, is really a file from the work node's perspective. So "chunk-level" is not listed here.

the same compressor, the compressed file would be 16MB+4KB*1K = 20MB. Therefore we would roughly spend (20MB-16.004MB)/16.004MB ≈ 25% more space in block-level compression than the file-level one.

Virtual chunks (VC) aim to better employ existing compression algorithms in parallel and distributed file systems, and eventually to improve the I/O performance of random data accesses in scientific applications and high-performance computing (HPC) systems. Virtual chunks do not break the original file into physical chunks or blocks, but append a small number of references to the end of file. Each of these references points to a specific block that is considered as a boundary of the virtual chunk. Because the physical entirety (or, continuity of blocks) of the original file is retained, the compression overhead and compression ratio keep comparable to those of file-level compression. With these additional references, a random file access need not decompress the entire file from the beginning, but could arbitrarily jump onto a reference close to the requested data and start the decompression from there. Therefore virtual chunks help to achieve the best of both file- and block-level compressions: high compression ratio and efficient random access.

6.2 Virtual Chunks

To make matters more concrete, we illustrate how virtual chunks work with an XOR-based delta compression [20] that is applied to parallel scientific applications. The idea of XOR-based delta compression is straightforward: calculating the XOR difference between every pair of adjacent data entries in the input file, so that only the very first data entry needs to be stored together with the XOR differences. This XOR compression proves to be highly effective for scientific data like climate temperatures, because the large volume of numerical values change marginally in the neighboring spatial and temporal area. Therefore, storing the large number of small XOR differences instead of the original data entries could significantly shrink the size of the compressed file.

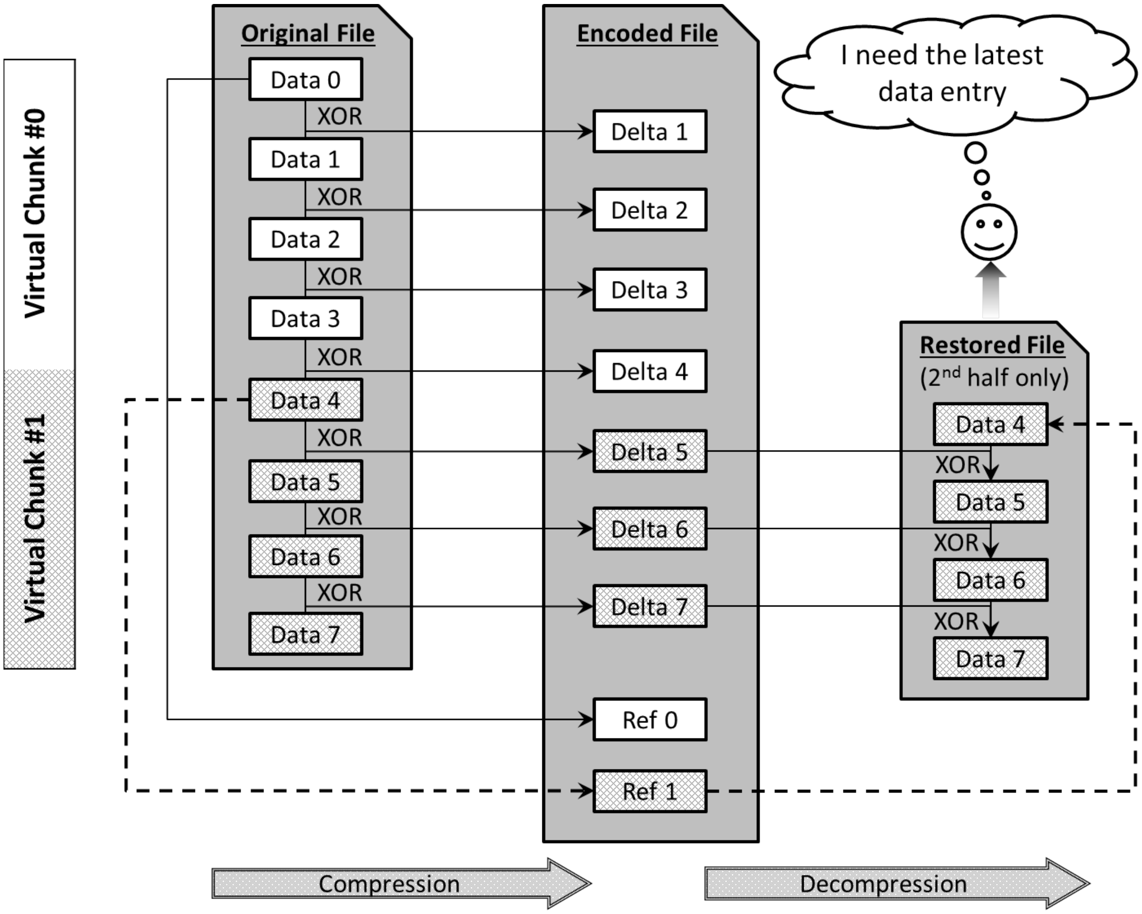


Figure 6.1. Compression and decompression with two virtual chunks

Figure 6.1 shows an original file of eight data entries, and two references to Data 0 and Data 4, respectively. That is, we have two virtual chunks of Data 0 – 3 and Data 4 – 7, respectively. In the compressed file, we store seven deltas and two references. When users need to read Data 7, we first copy the nearest upper reference (Ref 1 in this case) to the beginning of the restored file, then incrementally XOR the restored data and the deltas, until we reach the end position of the requested data. In this example, we roughly save half of the I/O time during the random file read by avoiding reading and decompressing the first half of the file.

For clear presentation of the following algorithms to be discussed, we assume the original file data can be represented as a list $D = \langle d_1, d_2, \ldots, d_n \rangle$. Since there are n data entries, we have $n-1$ encoded data, denoted by the list $X = \langle x_1, x_2, \ldots, x_{n-1} \rangle$ where $x_i = d_i$ XOR d_{i+1} for $1 \leq i \leq n-1$. We assume there are k references (i.e. original data entries) that the k virtual chunks start with. The k references are represented by a list $D' = \langle d_{c_1}, d_{c_2}, \ldots, d_{c_k} \rangle$, where for any $1 \leq i \leq k-1$ we have $c_i \leq c_{i+1}$. Notice that we need $c_1 = 1$, because it is the basis from where the XOR could be applied to the original data D. We define $L = \frac{n}{k}$, the length of a virtual chunk if the references are equidistant. The number in the square bracket [ ] after a list variable indicates the index of the scalar element. For example $D'[i]$ denotes the i^{th} reference in the reference list D'. This should not be confused with d_{c_i}, which represents the c_i^{th} element in the original data list D. The sublist starting at s and ending at t of a list D is represented as $D_{s,t}$.

6.2.1 Storing Virtual Chunks. We have considered two strategies on where to store the references: (1) put all references together (either in the beginning or in the end); (2) keep the reference in-place to indicate the boundary, i.e. spread out the references in the compressed file. Current design takes the first strategy that stores the references together at the end of the compressed file, as explained in the following.

The in-place references offer two limited benefits. Firstly, it saves space of $(k-1)$ encoded data entries (recall that k is the total number of references). For example, Delta 4 would not be needed in Figure 6.1. Secondly, it avoids the computation on locating the lowest upper reference at the end of the compressed file. For the first benefit, the space saving is insignificant because encoded data are typically much smaller than the original ones, not to mention this gain is factored by a relatively small number of reference $(k-1)$ comparing to the total number of data entries (n). The second benefit on saving the computation time is also limited because the CPU time on locating the reference is marginal compared to compressing the data entries.

The drawback of the in-place method is, even though not so obvious, critical: it introduces significant overhead when decompressing a large portion of data spanning over multiple logical chunks. To see this, let us imagine in Figure 6.1 that Ref 0 is above Delta 1 and Ref 1 is in the place of Delta 4. If the user requests the entire file, then the file system needs to read two raw data entries: Ref 0 (i.e. Data 0) and Ref 1 (i.e. Data 4). Note that Data 0 and Data 4 are original data entries, and are typically much larger than the deltas. Thus, reading these in-place references would take significantly more time than reading the deltas, especially when the requested data include a large number of virtual chunks. This issue does not exist in our current design where all references are stored together at the end of file: the user only needs to retrieve one reference (i.e. Ref 0 in this case).

6.2.2 Compression with Virtual Chunks. We use `encode()` (or `decode()`) applicable to two neighboring data entries to represent some compression (or decompression) algorithms to the file data. Certainly, it is not always true that the compression algorithm deals with two neighboring data entries; we only take this assumption for clear representation, and it would not affect the validity of the algorithms or the analysis that follows.

The procedure to compress a file with multiple references is described in Algorithm 0. The first phase of the virtual-chunk compression is to encode the original data entries of the original file, as shown in Lines 1–3. The second phase appends k references to the end of the compressed file, as shown in Lines 4–6.

The time complexity of Algorithm 0 is $O(n)$. Lines 1–3 obviously take $O(n)$ to compress the file. Lines 4–6 are also bounded by $O(n)$ since there cannot be more than n references in the procedure.

6.2.3 Optimal Number of References. This section answers this question:

Algorithm 6.1 VC Compress

```
Input: The original data $D = \langle d_1, \cdots, d_n \rangle$
Output: The encoded data $X$, and the reference list $D'$
for (int i = 1; i < n; i++) do
    $X[i] \leftarrow$ encode($d_i$, $d_{i+1}$)
end for
for (int j = 1; j < k; j++) do
    $D'[j] \leftarrow D[1 + (j - 1) * L]$
end for
```

how many references should we append to the compressed file, in order to maximize end-to-end I/O performance?

In general, more references consume more storage space, implying longer time to write the compressed data to storage. As an extreme example, making a reference to each data entry of the original file is not a good idea: the resulted compressed file is actually larger than the original file. On the other hand, however, more references yield a better chance of a closer lowest upper reference from the requested data, which in turn speeds up the decompression for random accesses. Thus, we want to find the number of references that has a good balance between compression and decompression, and ultimately achieves the minimal overall time.

Despite many possible access patterns and scenarios, in this chapter we are particularly interested in finding the number of references that results in the minimal I/O time in the worst case: for data write, the entire file is compressed and written to the disk; for data read, the last data entry is requested. That is, the decompression starts from the beginning of the file and processes until the last data entry. The following analysis is focused on this scenario, and assumes the references are equidistant.

Table 6.1. Virtual Chunk Parameters

Variable	Description
B_r	Read Bandwidth
B_w	Write Bandwidth
W_i	Weight of Input
W_o	Weight of Output
S	Original File Size
R	Compression Ratio
D	Computational Time of Decompression

A few more parameters for the analysis are listed in Table 6.1. We denote the read and the write bandwidth for the underlying file system by B_r and B_w, respectively. Different weights are assigned to input W_i and output W_o to reflect the access patterns. For example if a file is written once and then read for 10 times in an application, then it makes sense to assign more weights to the file read (W_i) than the file write (W_o). S indicates the size of the original file to be compressed. R is the compression ratio, so the compressed file size is $\frac{S}{R}$. D denotes the computational time spent on decompressing the requested data, which should be distinguished from the overall decompression time (D plus the I/O time).

The overhead introduced by additional references during compression is as follows. The baseline is when the file is applied with the conventional compression with a single reference. When comparing both cases, we need to apply the same compression algorithm to be applied on the same set of data. Therefore, the computational time should be unchanged regardless of the number of references appended to the compressed file. So the overall time difference really comes from the I/O time of writing different number of references.

Let T_c indicate the time difference between multiple references and a single reference, we have

$$T_c = \frac{(k-1) \cdot S \cdot W_o}{n \cdot B_w}$$

Similarly, to calculate the potential gain during decompression with multiple references, T_d indicating the time difference in decompression between multiple references and a single reference, is calculated as follows:

$$T_d = \frac{(k-1) \cdot S \cdot W_i}{k \cdot R \cdot B_r} + \frac{(k-1) \cdot D \cdot W_i}{k}$$

The first term of the above equation represents the time difference on the I/O part, and the second term represents the computational part.

To minimize the overall end-to-end I/O time, we want to maximize the following function (i.e. gain minus cost):

$$F(k) = T_d - T_c$$

Note that the I/O time is from the client's (or, user's) perspective. Technically, it includes both the computational and I/O time of the (de)compression. By taking the derivative on k (suppose $\hat{k}$ is continuous) and solving the following equation

$$\frac{d}{d\hat{k}}(F(\hat{k})) = \frac{S \cdot W_i}{R \cdot B_r \cdot \hat{k}^2} + \frac{D \cdot W_i}{\hat{k}^2} - \frac{S \cdot W_o}{B_w \cdot n} = 0,$$

we have

$$\hat{k} = \sqrt{n \cdot \frac{B_w}{B_r} \cdot \frac{W_i}{W_o} \cdot (\frac{1}{R} + \frac{D \cdot B_r}{S})}$$

To make sure $\hat{k}$ reaches the global maximum, we can take the second-order derivative on $\hat{k}$:

$$\frac{d^2}{d\hat{k}^2}(F(\hat{k})) = -\frac{S \cdot W_i}{R \cdot B_r \cdot \hat{k}^3} - \frac{D \cdot W_i}{\hat{k}^3} < 0$$

since all parameters are positive real numbers. Because the second-order derivative is always negative, we are guaranteed that the local optimal $\hat{k}$ is really a global maximum.

Since k is an integer, the optimal k is given as:

$$\arg\max_{k} F(k) = \begin{cases} \lfloor \hat{k} \rfloor & \text{if } F(\lfloor \hat{k} \rfloor) > F(\lceil \hat{k} \rceil) \\ \lceil \hat{k} \rceil & \text{otherwise} \end{cases}$$

Therefore the optimal number of references k_{opt} is:

$$k_{opt} = \begin{cases} \lfloor \hat{k} \rfloor & \text{if } F(\lfloor \hat{k} \rfloor) > F(\lceil \hat{k} \rceil) \\ \lceil \hat{k} \rceil & \text{otherwise} \end{cases} \tag{6.1}$$

where

$$\hat{k} = \sqrt{n \cdot \frac{B_w}{B_r} \cdot \frac{W_i}{W_o} \cdot (\frac{1}{R} + \frac{D \cdot B_r}{S})} \tag{6.2}$$

and

$$F(x) = \frac{(x-1) \cdot S \cdot W_i}{x \cdot R \cdot B_r} + \frac{(x-1) \cdot D \cdot W_i}{x} - \frac{(x-1) \cdot S \cdot W_o}{n \cdot B_w}$$

Note that the last term $\frac{D \cdot B_r}{S}$ in Equation 6.2 really says the ratio of D over $\frac{S}{B_r}$. That is, the ratio of the computational time over the I/O time. If we assume the computational portion during decompression is significantly smaller than the I/O time (i.e. $\frac{D \cdot B_r}{S} \approx 0$), the compression ratio is not extremely high (i.e. $\frac{1}{R} \approx 1$), the read and write throughput are comparable (i.e. $\frac{B_w}{B_r} \approx 1$), and the input and output weight are comparable (i.e. $\frac{W_i}{W_o} \approx 1$), then a simplified version of Equation 6.2 can be stated as:

$$\hat{k} = \sqrt{n} \tag{6.3}$$

suggesting that the optimal number of references be roughly the squared root of the total number of data entries.

6.2.4 Random Read. This section presents the decompression procedure when a request of random read comes in. Before that, we describe a subroutine that is useful for the decompression procedure and more procedures to be discussed in later sections. The subroutine is presented in Algorithm 6.2, called *DecompList*. It is not surprising for this algorithm to have inputs such as encoded data X, and the starting and ending positions (s and t) of the requested range, while the latest reference no later than s (i.e. $d_{s'}$) might be less intuitive. In fact, $d_{s'}$ is not supposed to be specified from a direct input, but calculated in an ad-hoc manner for different scenarios. We will see this in the complete procedure for random read later in this section.

Algorithm 6.2 DecompList

Input: The start position s, the end position t, the latest reference no later than s as $d_{s'}$, the encoded data list $X = \langle x_1, x_2, \ldots, x_{n-1} \rangle$

Output: The original data between s and t as $D_{s,t}$

```
1: $prev \leftarrow d_{s'}$
2: for $i = s'$ to $t$ do
3:     if $i \geq s$ then
4:         $D_{s,t}[i - s] \leftarrow prev$
5:     end if
6:     $prev \leftarrow$ encode($prev$, $x_i$)
7: end for
```

In Algorithm 6.2, Line 1 stores the reference in a temporary variable as a base value. Then Lines 2 – 7 decompress the data by increasingly applying the decode function between the previous original value and the current encoded value. If the decompressed value lands in the requested range, it is also stored in the return list.

Now we are ready to describe the random read procedure to read an arbitrary data entry from the compressed file. Recall that in static virtual chunks, all reference are equidistant. Therefore, given the start position s we could calculate its closest and latest reference index $s' = LastRef(s)$ where :

$$LastRef(x) \leftarrow \begin{cases} \frac{x}{L} + 1 & \text{if } 0 \neq x \texttt{ MOD } L \\ \\ \frac{x}{L} & \text{otherwise} \end{cases} \tag{6.4}$$

So we only need to plug Equation 6.4 to Algorithm 6.2. Also note that we only use Algorithm 6.2 to retrieve a single data point, therefore we can set $t = s$ in the procedure.

The time complexity of random read is $O(L)$, since it needs to decompress as much as a virtual chunk to retrieve the requested data entry. If a batch of read requests comes in, a preprocessing step (e.g. sorting the positions to be read) can be applied so that decompressing a virtual chunk would serve multiple requests.

It should be clear that the above discussion assumes the references are equidistant, i.e. static virtual chunks. And that is why we could easily calculate s' by Equation 6.4.

6.2.5 Random Write. The procedure of random write (i.e. modify a random data entry) is more complicated than the case of random read. In fact, the first step of random write is to locate the affected virtual chunk, which shares a similar procedure of random read. Then the original value of the to-be-modified data entry is restored from the starting reference of the virtual chunk. In general, two encoded values need to be updated: the requested data entry and the one after it. There are two trivial cases when the updated data entry is the first or the last. If the requested data entry is the first one of the file, we only need to update the first reference and the encoded data after it. This is because the first data entry always serves as the first reference as well. If the requested data entry is the last one of the file, then we just load the last reference and decode the virtual chunk till the end of file. In the following discussion, we consider the general case excluding the above two scenarios. Note that, if the requested data entry happens to be a reference, it needs to be updated as well with the new value.

Algorithm 6.3 VC Write

Input: The index of the data entry to be modified q, the new value v, encoded data $X = \langle x_1, x_2, \cdots, x_{n-1} \rangle$, and the reference list $D' = \langle d_1, d_2, \cdots, d_k \rangle$

Output: Modified X

```
1: $s' \leftarrow LastRef(q)$
2: $\langle d_{q-1}, d_q, d_{q+1} \rangle \leftarrow DecompList(q-1, q+1, d_{s'}, X)$
3: $x_{q-1} \leftarrow encode(d_{q-1}, v)$
4: $x_q \leftarrow encode(v, d_{q+1})$
5: if $0 = (q-1)$ MOD $L$ then
6:     $D'[\frac{q}{L} + 1] \leftarrow v$
7: end if
```

The procedure of updating an arbitrary data point is described in Algorithm 6.3. The latest reference no later than the updated position q is calculated in Line 1, per Equation 6.4. Then Line 2 reuses Algorithm 6.2 to restore three original data entries in the original file. They include the data entry to be modified, and the two adjacent ones to it. Line 3 and Line 4 re-compress this range with the new value v. Lines 5 – 7 check if the modified value happens to be one of the references. If so, the reference is updated as well.

The time complexity is $O(L)$, since all lines take constant time, except that Line 2 takes $O(L)$. If there are multiple update requests to the file, i.e. batch of requests, we can sort the requests so that one single pass of restoring a virtual chunk could potentially update multiple data entries being requested.

6.2.6 Updating VC. If the access pattern does not follow the uniform distribution, and this information is exposed to users, then it makes sense to specify more references (i.e. finer granularity of virtual chunks) for the subset that is more frequently accessed. This is because more references make random accesses more efficient with a shorter distance (and less computation) from the closest reference, in general. The assumption of equidistant reference, thus, does not hold any more in the following discussion.

While a self-adjustable mechanism to update the reference positions is ongoing at this point, this chapter expects that the users would specify the distribution of the reference density in a configuration file, or more likely a rule such as a decay function [38]. For those users who are really familiar with their data, a function that adjusts any particular range of data with an arbitrary number of references is also desirable. That is, the second type of users would need to access a lower level of reference manipulations. Note that, the specifications and distributions required by the first type of users could be implemented by the functions for the second type of users. Therefore, we decide to expose the interface to allow users (i.e. the second type of users) to control the finer granularity of reference adjustment. It should be fairly straightforward for the first type of users to meet their needs by extending the provided interfaces.

Before discussing the procedure to update the references, we will first describe some auxiliary functions. The `FindRef` function finds the latest reference no later than the given data index. It takes two inputs: the list of references and a data index, then applies a binary search to return the closest reference that is not later than the input data index. Since this only trivially extends the standard binary search, we do not give the formal algorithm in this chapter. This procedure takes $O(\log k)$ time, where k is the list length of all the references. Then we define the `FindSublist` function that extends `FindRef` with two input data indexes s and t such that $1 \leq s \leq t \leq n$ and the return value as a sublist $D'_{s',t'}$ such that $s' = \texttt{FindRef}(D', s)$ and $t' = \texttt{FindRef}(D', t)$.

To make the virtual chunks adjustable we will introduce the `RefUpdate` procedure that allows users to specify a linear transform of the existing virtual chunks within a particular range. Not surprisingly, this procedure requires more computation and possibly more parsing time if the updating rules are specified in a user-defined configuration file. This tradeoff between performance and flexibility is highly application-dependent. Thanks to the `RefUpdate` procedure, it is relatively straightforward to extend the file operations described in the static virtual chunks to their dynamic parities.

We assume there are m disjoint subsets of D that will be updated with a new number of references. Users are expected to specify the following parameters: the starting and ending position of a subset (s_i, t_i), as well as the coefficients in the linear transform α_i and β_i, where $1 \leq i \leq m$. Note that both s_i and t_i are the distances from the beginning of D where $1 \leq s_i < t_i \leq n$.

In the updating procedure, a sublist of D', namely $D'' = \langle d_{c^i_b}, d_{c^i_{b+1}}, \ldots, d_{c^i_e} \rangle$ is affected when the granularity within this range is updated. Note that c^i_b should be the immediate precedent of s_i, and c^i_e should be the immediate precedent of e_i. That is, there does not exist such a b' that $b' > b$ and $c^i_{b'} \leq s_i$; and there does not exist such an e' that $e' > e$ and $c^i_{e'} \leq t_i$. α is a float number meaning that we want α times as many as the original number of virtual chunks between s_i and t_i. β is the constant adjustment in the linear transform. We also compute a sublist D^* where $|D^*_i| = \alpha_i |D''_i| + \beta_i$, which will replace D''_i and then be inserted into D'. The list $\overline{D_i} = d_{c^i_b}, d_{c^i_b+1}, \ldots, d_{c^i_e}$ (i.e. another sublist of D) is also needed for the computation.

The procedure is presented in Algorithm 0. Lines 1 – 5 compute affected sublists of references D''_i and original data entries $\overline{D_i}$ for all the m requested updates. Lines 6 – 13 calculate the values of the affected or newly added references in the compressed data. Line 14 updates the reference values.

Algorithm 6.4 RefUpdate

Input: For $1 \leq i \leq m$, $(s_i,\, t_i)$, α_i, β_i, D', X.

Output: Modified D'.

```
for $i = 1$ to $m$ do
    $D_i'' \leftarrow \texttt{FindSublist}(s_i, t_i, D')$
    $s_i' \leftarrow \texttt{FindRef}(D', s_i)$
    $\overline{D_i} \leftarrow \texttt{DecompList}(s_i, t_i, d_{s_i'}, X)$
end for
for $i = 1$ to $m$ do
    for $j = 0$ to $|\overline{D_i}| - 1$ do
        $l \leftarrow \left\lfloor \frac{|\overline{D_i}|}{\alpha_i |D_i''| + \beta_i} \right\rfloor$
        if $0 = j \texttt{ MOD } l$ then
            Add $d_{c_b^i + j}$ to $D_i^*$.
        end if
    end for
end for
Replace $D_i''$ by $D_i^*$ in $D'$ for $1 \leq i \leq m$.
```

The time complexity of Algorithm 0 is as follows. In m iterations, Lines 2 – 3 take $O(m \log k)$ in total. Line 4 takes at most $O(n)$ in m iterations because each interval (s_i, t_i) is disjoint to others. Similarly, Lines 6 – 13 take at most $O(n)$ in m iterations. Line 14 also takes at most $O(n)$. So the overall time complexity is $O(m \log k + n)$. In practice, Algorithm 0 would not be frequently called, because users normally do not need to adjust the virtual chunk granularity for every change to the data.

Once the references are updated, we cannot simply locate the reference by dividing the total number of data entries by the size of the virtual chunk as we did in the static case. However, with the help of the `FindRef` function, we can still retrieve the closest reference before the given data index by a binary search. For example, Line 1 of Algorithm 6.3 needs to be replaced by

$$s' \leftarrow \texttt{FindRef}(D', s)$$

if the references are not equidistant. Similarly, the `LastRef` function call in the random read procedure needs to be replaced by `FindRef`. The time complexity of the dynamic-reference algorithms (i.e. random read and random write) is $O(L' + \log k)$, where L' indicates the size of the affected virtual chunk (not equidistant anymore) and $O(\log k)$ represents the time of `FindRef`.

If a subset is frequently accessed, the rule of thumb is to increase the reference density of this area. In this case, L' becomes small to indicate such fine granularity. It then implies that the overall complexity would not become significantly high even for frequent reference updates. So the random read and random write are still flexibly and efficiently maintained without much overhead compared to the static case. We will provide a detailed analysis on the potential I/O improvement by paying such overhead.

6.2.7 I/O Improvement from Dynamic VC. This section analyzes the I/O benefit from dynamic virtual chunks. As discussed before, updating static virtual chunks into dynamic ones introduces the overhead of adjusting the references (Algorithm 0). The goal of paying this overhead is to place more references to a frequently accessed subset of data.

To make a clear presentation, we make the following assumptions. Suppose n is dividable by k (i.e. $n \texttt{ MOD } k = 0$), so that in the static setting all k virtual chunks are of the same size $L = \frac{n}{k}$. We assume there are two updates to the static references on $(1, \frac{n}{c})$ and $(\frac{n}{c} + 1, n)$, respectively, where c is a integer to control the boundary between the two portions. This c variable is supposed to be significantly larger than one (i.e. $c \gg 1$), and $\frac{1}{c} \ll \frac{c-1}{c}$. The first update has parameters α_1, β_1, $s_1 = 1, t_1 = \frac{n}{c}$, and the second one has parameters α_2, β_2, $s_2 = \frac{n}{c} + 1, t_2 = n$. To make the dynamic case comparable to the static virtual chunk, the total number of virtual chunks after both updates is kept the same:

$$\alpha_1 \cdot \frac{k}{c} + \beta_1 + \alpha_2 \cdot \frac{(c-1) \cdot k}{c} + \beta_2 = k$$

On the other hand, we assume the smaller portion on $(1, \frac{n}{c})$ has a finer granularity of virtual chunks:

$$\alpha_1 \cdot \frac{k}{c} + \beta_1 \gg \alpha_2 \cdot \frac{(c-1) \cdot k}{c} + \beta_2$$

Finally, we assume there are f consecutive random I/Os to be applied to the portion on $(1, \frac{n}{c})$.

Without the two reference updates, the cost of f I/Os is simply $O(f \cdot L)$, since each I/O can take up to $O(L)$. Now we consider the dynamic case. By our previous analysis (Algorithm 0), it takes $O(m \cdot \log k + n)$ to complete m updates if the references are already updated for dynamic virtual chunks. In this scenario, we only have two updates ($m = 2$) and the virtual chunks before the updates are static ($\log k \to 1$) since Line 3 of Algorithm 0 can be directly calculated by `LastRef`, and the total cost of the updates is just $O(n)$. Thus the total cost of f I/Os in dynamic virtual chunks is

$$O(n + f \cdot \log k + f \cdot L \cdot \frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1})$$

Note that the cost of static virtual chunks is $O(f \cdot L)$. Therefore, to make the dynamic reference beneficial in terms of the overall I/O performance, we need

$$f \cdot L > n + f \cdot \log k + f \cdot L \frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1}$$

or

$$L \cdot (1 - \frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1}) - \log k > \frac{n}{f} \tag{6.5}$$

In practice, the condition in Equation 6.5 is easy to satisfy. On the left hand side of Equation 6.5, since the number of references is significantly increased on $(1, \frac{n}{c})$, $\frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1}$ is significantly smaller than 1 so that $L \cdot (1 - \frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1})$ is close to L. Also note that $\log k$ is a lot smaller than L since it is the logarithmic of the reference number. On the right hand side, because we assume the portion on $(1, \frac{n}{c})$ is frequently accessed, easily making $\frac{n}{f}$ smaller than the left hand side.

6.3 Experiment Results

We have implemented a user-level compression middleware for GPFS [172] with the FUSE framework [59]. The compression logic is implemented in the *vc_write()* interface, which is the handler for catching the write system calls. *vc_write()* compresses the raw data, caches it in the memory if possible, and writes the compressed data into GPFS. The decompression logic is implemented in the *vc_read()* interface, similarly. When a read request comes in, this function loads the compressed data (either from the cache or the disk) into memory, applies the decompression algorithm to the compressed data, and passes the result to the end users.

The virtual chunk middleware is deployed on each compute node as a mount point that refers to the remote GPFS file system. This architecture enables a high possibility of reusing the decompressed data, since the decompressed data are cached in the local node. Moreover, because the original compressed file is split into many logical chunks each of which can be decompressed independently, it allows a more flexible memory caching mechanism and parallel processing of these logical chunks. We have implemented a LRU replacement policy for caching the intermediate data.

We have also integrated virtual chunks into the FusionFS [225] distributed file system. The key feature of FusionFS is to fully exploit the available resources and avoid any centralized component. That is, each participating node plays three roles at the same time: client, metadata server, and data server. Each node is able to pull the global view of all the available files by the single namespace implemented with a distributed hash table [101], even though the metadata is physically distributed on all the nodes. Each node stores parts of the entire metadata and data at its local storage. Although both metadata and data are fully distributed on all nodes, the local metadata and data on the same node are completely decoupled: the local data may or may not be described by the local metadata. By decoupling metadata and data, we are able to apply flexible strategies on metadata management and data I/Os.

On each compute node, a virtual chunk component is deployed on top of the data I/O implementation in FusionFS. FusionFS itself has employed FUSE to support POSIX, so there is no need for VC to implement FUSE interfaces again. Instead, VC is implemented in the *fusionfs_write()* and the *fusionfs_read()* interfaces. Although the compression is implemented in the *fusionfs_write()* interface, the compressed file is not persisted into the hard disk until the file is closed. This approach can aggregate the small blocks into larger ones, and reduce the number of I/Os to improve the end-to-end time. In some scenarios, users are more concerned for the high availability rather than the compressing time. In that case, a *fsync()* could be called to the (partially) compressed data to ensure these data are available at the persistent storage in a timely manner, so that other processes or nodes could start processing them.

The remainder of this section answers following questions:

1. How does the number of VC affect the compression ratio and sequential I/O time?
2. How does VC, as a middleware, improve the GPFS [172] I/O throughput?
3. How does VC, as a built-in component, help to improve the I/O throughput of FusionFS [225]?

All experiments were repeated at least five times, or until results became stable (i.e. within 5% margin of error); the reported numbers are the average of all runs.

6.3.1 Compression Ratio. We show how virtual chunks affect the compression ratio on the Global Cloud Resolving Model (GCRM) data [60]. GCRM consists of single-precision float data of temperatures to analyze cloud's influence on the atmosphere and the global climate. In our experiment there are totally $n = 3.2$ million data entries to be compressed with the aforementioned XOR compressor. Each data entry comprises a row of 80 single-precision floats. Note that based on our previous analysis, the optimal number of references should be set roughly to $\sqrt{n} \approx 1,789$ (Equation 6.3). Thus we tested up to 2,000 references, a bit more than the theoretical optimum.

From 1 to 2,000 references, the compression ratio change is reported in Table 6.2, together with the overall wall time of the compression. As expected, the compression ratio decreases when more references are appended. However, the degradation of compression ratio is almost negligible: within 0.002 between 1 reference and

Table 6.2. Overhead of Additional References

Number of References	Compression Ratio	Wall Time (second)
1	1.4929	415.40
400	1.4926	415.47
800	1.4923	415.54
1200	1.4921	415.62
1600	1.4918	415.69
2000	1.4915	415.76

2000 references. These small changes to the compression ratios then imply negligible differences of the wall time also: within sub-seconds out of minutes. Thus, this experiment demonstrates that adding a reasonable number of additional references, guided by our analysis, only introduces negligible overhead to the compression process.

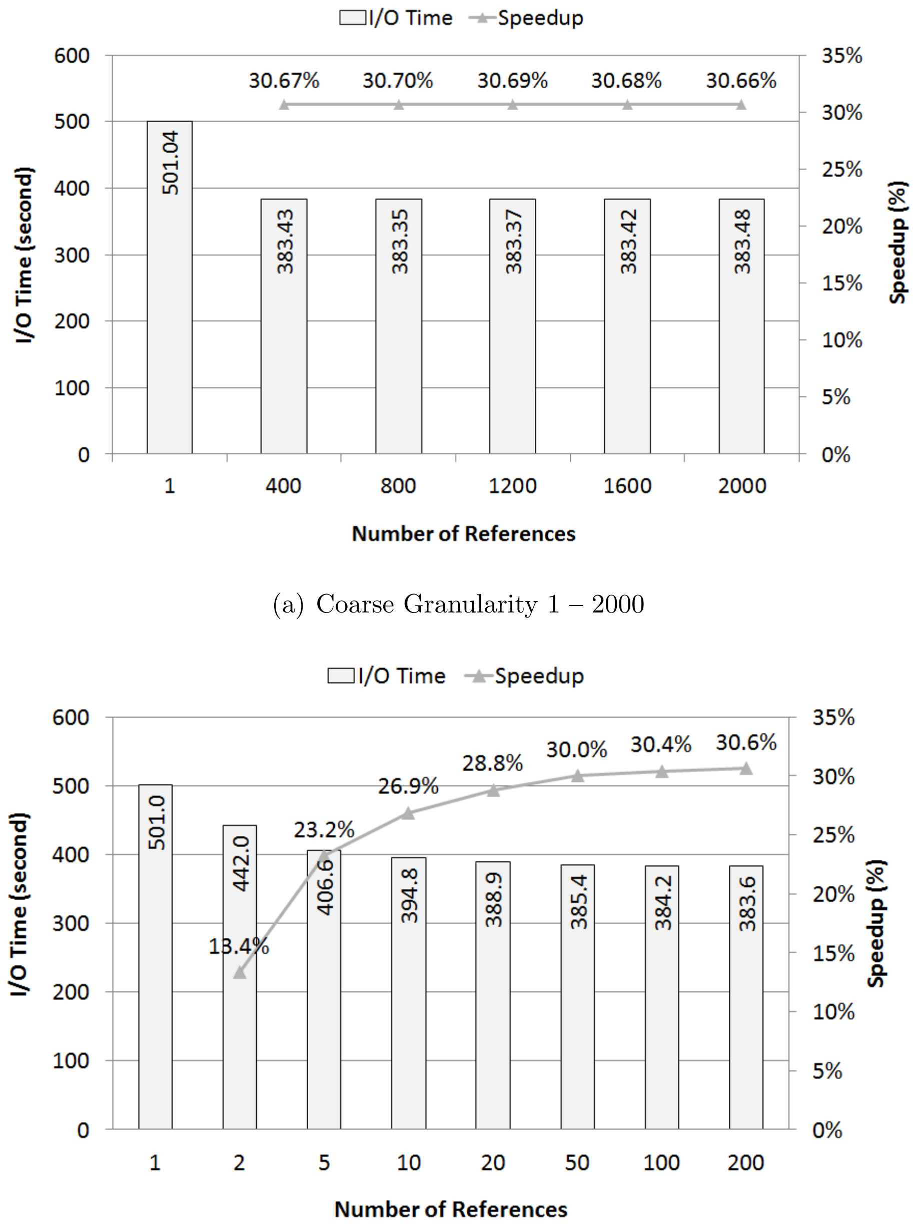
The reason of the negligible overhead is in fact due to Equation 6.2, or Equation 6.3 as a simplified version. The total number of data entries is about quadratic to the optimal number of references, making the cost of processing the additional references only marginal to the overall compression procedure, particularly when the data size is large.

6.3.2 GPFS Middleware. We deployed the virtual chunk middleware on 1,024 cores (256 physical nodes) pointing to a 128-nodes GPFS [172] file system on Intrepid [79], an IBM Blue Gene/P supercomputer at Argonne National Laboratory. Each Intrepid compute node has a quad-core PowerPC 450 processor (850MHz) and 2GB of RAM. The dataset is 244.25GB of the GCRM [60] climate data.

Since virtual chunk is implemented with FUSE [59] that adds extra context switches when making I/O system calls, we need to know how much overhead is induced by FUSE. To measure the impact of this overhead, the GCRM dataset is written to the original GPFS and the GPFS+FUSE file system (without virtual chunks), respectively. The difference is within 2.2%, which could be best explained by the fact that in parallel file systems the bottleneck is on the networking rather than the latency and bandwidth of the local disks. Since the FUSE overhead on GPFS is smaller than 5%, we will not distinguish both setups (original GPFS and FUSE+GPFS) in the following discussion.

We tested the virtual chunk middleware on GPFS with two routine workloads: (1) the archival (i.e. write with compression) of all the available data; and (2) the retrieval (i.e. read with decompression) of the latest temperature, regarded as the worst-case scenario. The I/O time, as well as the speedup over the baseline of single-reference compression, is reported in Figure 6.2(a). We observe that multiple references ($400 - 2000$) significantly reduce the original I/O time from 501s to 383s, and reach the peak performance at 800-references with 31% (1.3X) improvement.

An interesting observation from Figure 6.2(a) is that, the performance sensitivity to the number of references near the optimal k_{opt} is relatively low. The optimal number of references seems to be 800 (the shortest time: 383.35 seconds), but the difference across 400-2000 references is marginal, only within sub-seconds. This phenomenon is because that beyond a few hundreds of references, the GCRM data set has reached a fine enough granularity of virtual chunks that could be efficiently decompressed. To justify this, we re-run the experiment with finer granularity from 1 to 200 references as reported in Figure 6.2(b). As expected, the improvement over 1–200 references is more significant than between 400 and 2000. This experiment also indicates that, we could achieve a near-optimal (within 1%) performance (30.0% speedup at $k = 50$ vs 30.70% at $k = 800$) with only $\frac{50}{800} = 6.25\%$ cost of additional ref-



(a) Coarse Granularity 1 – 2000

(b) Fine Granularity 1 – 200

Figure 6.2. I/O time with virtual chunks in GPFS

erences. It thus implies that even fewer references than $\sqrt{n}$ could become significantly beneficial to the overall I/O performance.

To study the effect of virtual-chunk compression to real applications, we ran the MMAT application [20] that calculates the minimal, maximal, and average temperatures on the GCRM dataset. The breakdown of different portions is shown in Figure 6.3. Indeed, MMAT is a data-intensive application, as this is the application type where data compression is useful. So we can see that in vanilla GPFS 97% (176.13 out of 180.97 seconds) of the total runtime is on I/O. After applying the compression layer ($k = 800$), the I/O portion is significantly reduced from 176.13 to 118.02 seconds. Certainly this I/O improvement is not free, as there is 23.59 seconds overhead for the VC computation. The point is, this I/O time saving (i.e. 176.13 - 118.02 = 58.11 seconds) outweighs the VC overhead (23.59 seconds), resulting in 1.24X speedup on the overall execution time.

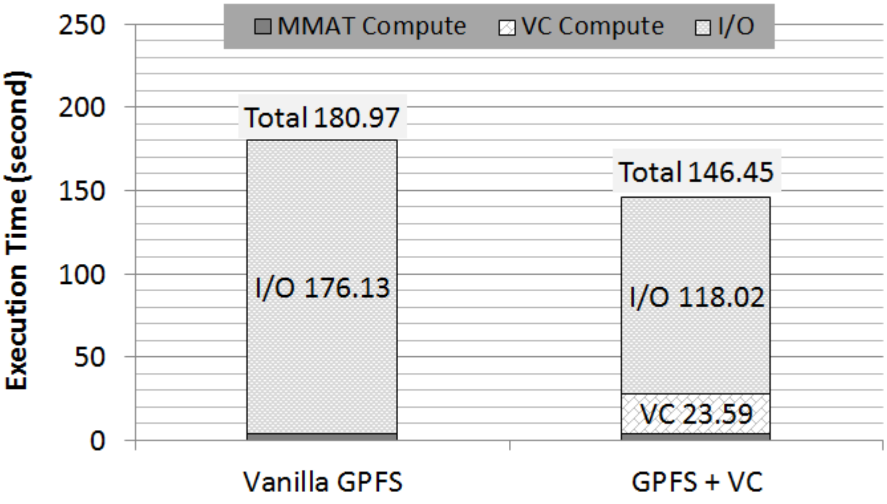


Figure 6.3. Execution time of the MMAT application

6.3.3 FusionFS Integration. We have deployed FusionFS integrated with virtual chunks to a 64-nodes Linux cluster at Illinois Institute of Technology. Each node has two Quad-Core AMD Opteron 2.3GHz processors with 8GB RAM and 1TB Seagate Barracuda hard drive. All nodes are interconnected with a 1Gbps Ethernet. Besides the GCRM [60] data, we also evaluated another popular data set Sloan Digital Sky Survey (SDSS [175]) that comprises a collection of astronomical data such as positions and brightness of hundreds of millions of celestial objects.

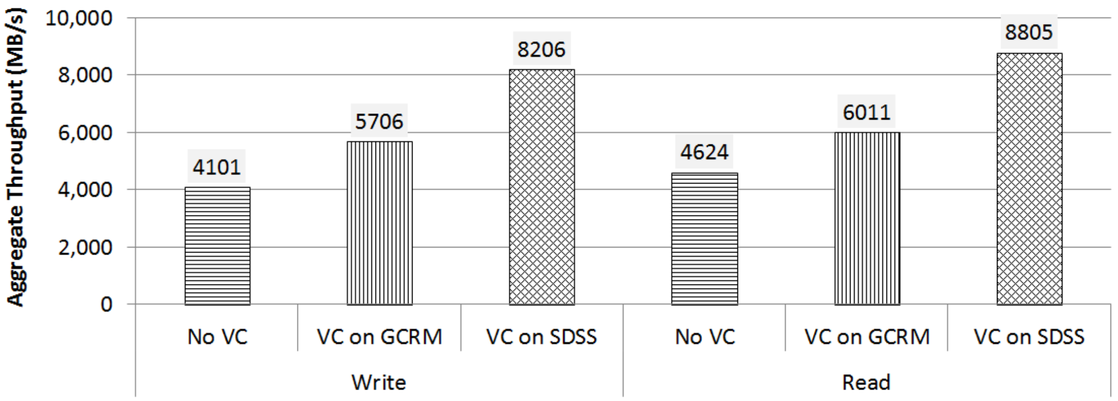


Figure 6.4. FusionFS throughput on GCRM and SDSS datasets

We illustrate how virtual chunks help FusionFS to improve the I/O throughput on both data sets in Figure 6.4. We do not vary k but set it to $\sqrt{n}$ when virtual chunk is enabled. Results show that both read and write throughput are significantly improved. Note that, the I/O throughput of SDSS is higher than GCRM, because the compression ratio of SDSS is 2.29, which is higher than GCRM's compression ratio 1.49. In particular, we observe up to 2X speedup when VC is enabled (SDSS write: 8206 vs. 4101).

6.3.4 Parameter Sensitivity of Dynamic Virtual Chunks. As discussed before, it is nontrivial to update the granularity (i.e. linear transform by α_i and β_i) of virtual chunks within a particular range (i.e. (s_i, t_i)). We will show quantitatively how costly this update computation is with respect to static virtual chunks. The experiments assume there is one update applied each time, so the subscripts of α_1, β_1, and (s_i, t_i) are not shown in the following discussion.

It should be clear that the cost in the following discussion is only for the reference update, and does not consider the benefit from consequent I/Os as discussed before. Therefore, even if the overhead is large e.g. 50%, this cost could be compensated by the I/O savings, say 60%, that achieves a better overall performance by 60%

- 50% = 10%.

The experimental setup is as follows. The files being evaluated are again the 244.25GB GCRM data [60]. After being compressed with 2,000 equidistant virtual chunks, the `RefUpdate` procedure is triggered to adjust the virtual chunks. The runtime of this procedure is compared to the time of compressing the data with static virtual chunks; the ratio of the update time over the compressing time is then considered as the cost (in %).

There are two dimensions to control the updating behavior: (1) the affected range length $(s - t + 1)$, and (2) the linear transform with α and β. Intuitively, a larger $(s-t+1)$ indicates more computation, since more references need to be updated within that range. This intuition also applies to the linear transform: larger α and β imply more references to be appended to the end of the compressed file. Note that, even though both α and β are considered as coefficients in the linear transform, it is sufficient to adjust α and set $\beta = 0$ to study the performance with respect to the density of references, or in other words the granularity of virtual chunks. Therefore in the following discussion, β is set to zero.

In order to study the combined effect of both the updated range and chunk granularity, we tune $\frac{s-t+1}{n}$ to be 0.2 – 1.0, and α to be 0.5 – 128 times of the original granularity. Figure 6.5 shows the cost from different parameter combinations. Unsurprisingly, the results confirm our previous intuition: the peak cost (61.15%) comes from the scenario where: (1) all the references are updated, i.e. $\frac{s-t+1}{n} = 1$, and (2) the granularity of virtual chunks is increased by 128 (the most) times. Similarly, the lowest cost (10.07%) occurs for $\frac{s-t+1}{n} = 0.2$ and $\alpha = 0.5$, the smallest values of both dimensions.

Now, we turn to discuss a more interesting observation from Figure 6.5: the

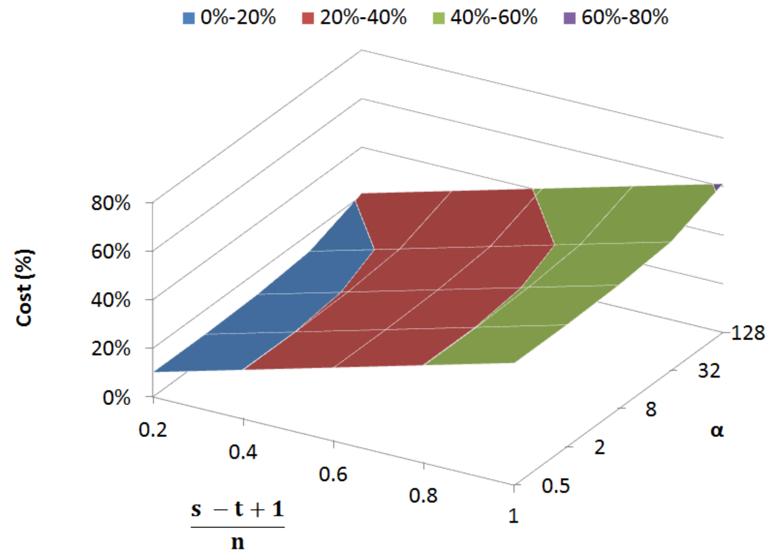


Figure 6.5. Parameter sensitivity of dynamic virtual chunks

updated range seems to have a more significant impact to the overhead, than α does. In particular, for all α's, the increased overhead looks strongly proportional to the increased $(s - t + 1)$ range, while the impact from α is less noticeable: increasing α from 0.5 to 128 only adds about 10% cost. To make this more obvious, we slice the 3-D surface on two dimensions when fixing $\frac{s-t+1}{n} = 0.5$ and $\alpha = 2$, as shown in Figure 6.6 and Figure 6.7, respectively.

Figure 6.6 clearly shows that changing α from 0.5 to 128 affects the cost by slightly less than 11%, with a fixed ratio between the updated range and the overall length ($\frac{s-t+1}{n} = 0.5$). This could be best explained by the fact that the number of references is roughly set to the square root of the number of data entries. Thus, even though the number of references (controlled by α and β) is significantly increased (256X from 0.5 to 128), the overall impact to the system performance is diluted by the small factor – the square root of the original scale.

In contrast to α, we observed a strong linearity between the cost and the

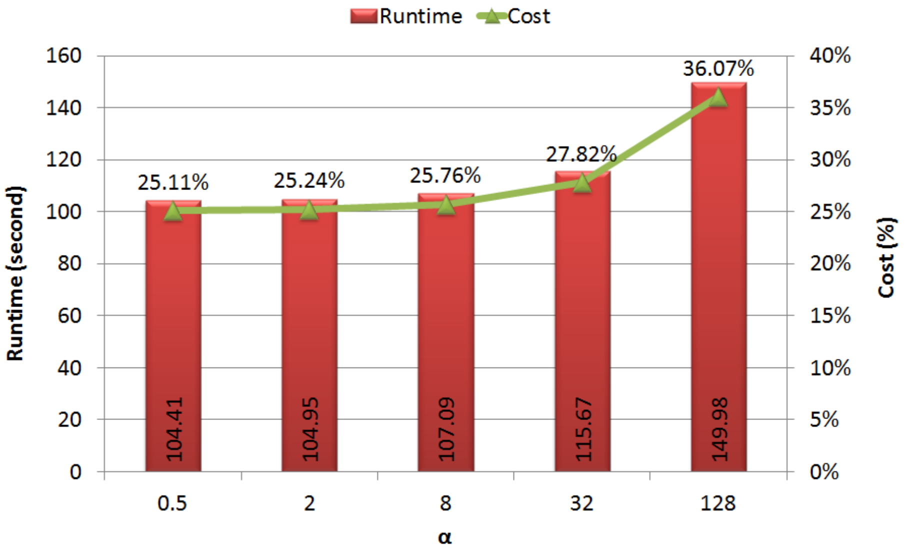


Figure 6.6. Parameter sensitivity with fixed range ratio = 0.5

updated range $(s - t + 1)$, as shown in Figure 6.7. The reason of this phenomenon is that all the encoded data within (s, t) need to be read into memory and then decoded to retrieve the new references. Therefore the cost of this procedure is highly dependent on the number of data within the range, and it is exactly why we see a strong linear relation between the overhead and the updated range.

6.4 Discussions and Limitations

6.4.1 Applicability. It should be clear that the proposed virtual chunk mechanism to be used in compressible storage systems is applicable only if the underlying compression format is splittable. A compressed file is splittable if it can be split into subsets and then be processed (e.g. decompressed) in parallel. Obviously, one key advantage of virtual chunks is to manipulate data in the arbitrary and logical subsets of the original file, which depends on this splittable feature. Without a splittable compression algorithm, the virtual chunk is not able to decompress itself. The XOR-based delta compression used through this chapter is clearly a splittable format. Popular compressors, such bzip2 [24] and LZO [110], are also splittable. Some

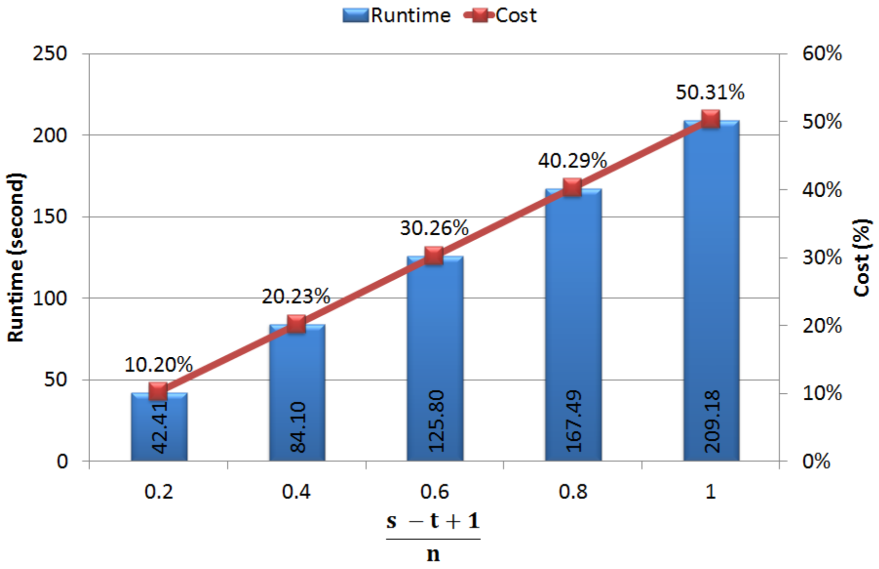


Figure 6.7. Parameter sensitivity with fixed $\alpha = 2$

non-splittable examples include Gzip [71] and Snappy [179].

It should also be noted that virtual chunks are not designed for general-purpose compression, but for highly compressible scientific data. This is why this study did not evaluate a virtual chunk version of general compressors (e.g. bzip2, LZO), since they are not designed for numerical data used in scientific applications.

6.4.2 Dynamic Virtual Chunks. If the access pattern does not follow the uniform distribution, and this information is exposed to users, then it makes sense to specify more references (i.e. finer granularity of virtual chunks) for the subset that is more frequently accessed. This is because more references make random accesses more efficiently with a shorter distance (and less computation) from the closest reference, in general. The assumption of equidistant reference, thus, would not hold any more in this case.

One intuitive solution to adjust the virtual chunk granularity is to ask users to specify where and how to update the reference. It implies that the users are expected to have a good understanding of their applications, such as I/O patterns. This is a reasonable assumption in some cases, for example if the application developers are the main users. Therefore, we expect that the users would specify the distribution of the reference density in a configuration file, or more likely a rule such as a decay function [38].

Nevertheless we believe it would be more desirable to have an autonomic mechanism to adjust the virtual chunks for those domain users without the technical expertise such as chemists, astronomers, and so on. This remains an open question to the community and a direction of our future work.

6.4.3 Data Insertion and Data Removal. We are not aware of much need for data insertion and data removal within a file in the context of HPC or scientific applications. By insertion, we mean a new data entry needs to be inserted into an arbitrary position of an existing compressed file. Similarly, by removal we mean an existing value at an arbitrary position needs to be removed. Nevertheless, it would make this work more complete if we supported efficient data insertion and data removal when enabling virtual chunks in storage compression.

A straightforward means to support this operation might treat a data removal as a special case of data writes with the new value as null. But then it would bring new challenges such as dealing with the "holes" within the file. We do not think either is a trivial problem, and would like to have more discussions with HPC researchers and domain scientists before investing in such features.

6.5 Summary

Conventional file- and block-level storage compression have shown their limits for scientific applications: file-level compression provides little support for random access, and block-level compression significantly degenerates the overall compression ratio due to the per-block compression overhead. This chapter introduces virtual chunks to support efficient random accesses to compressed scientific data while retaining the high compression ratio. Virtual chunks keep files' physical entirety, because they are referenced by pointers beyond the file end. The physical entirety helps to achieve a high compression ratio by avoiding the per-block compression overhead. The additional references take insignificant storage space and add negligible end-to-end I/O overhead. Virtual chunks enable efficient random accesses to arbitrary positions of the compressed data without decompressing the whole file. Procedures for manipulating virtual chunks are formulated, along with the analysis of optimal parameter setup. Evaluation demonstrates that virtual chunks improve scientific applications' I/O throughput by up to 2X speedup at large scale.

CHAPTER 7

GPU ENCODING

This chapter investigates if GPU technologies can speedup erasure coding to replace conventional file replication. Erasure coding offers high data reliability with less space, as it does not require full replicas but only smaller parities. The major critique for erasure coding, however, lies on its computational overhead, because the encoding and decoding process used to be extremely slow on conventional CPUs due to complex matrix computations. Nevertheless, today's GPUs are architected with massively concurrent computing cores that are good at SIMD applications such as matrix computations. To justify the feasibility of GPU-accelerated erasure coding, we build a GPU-accelerated erasure-coding-based distributed key-value store (Gest) from the ground up. Preliminary results were published in [220]. Experiment results show that Gest, when properly configured, achieves the same level of reliability as data replication, but with significantly higher space efficiency and I/O performance.

This work is orthogonal to previous studies that are focused on algorithms, protocols, and models for better manipulating replicas [127, 46, 18, 170, 92]: we propose to replace full-size replication by more space-efficient parity coding along with GPU acceleration. Although the idea is similar to Redundant Array of Independent Disks (RAID [147]), this work, to the best of our knowledge, is one of the pioneer studies that explore feasibility of taking advantages of both parity coding and GPU acceleration in distributed KVS. More specifically, instead of sending out full-size replicas, we split the value into smaller chunks, encode them with additional parities with GPUs, and disperse the encoded chunks onto remote nodes. This approach offers two advantages over the conventional full-size replication. First, the additional space required by the redundant data is greatly reduced. Second, the data parallelism is better exploited by the massive number of GPU cores.

While the coding overhead can be largely compensated by GPUs, one key challenge of the proposed approach is how to maintain the data locality of the scattered value (or, file) because the original value is split and encoded into smaller chunks that are physically stored on different nodes—the file-level locality from application's perspective is completely lost. To this end, we propose a locality-aware scheduling mechanism to directly assign the sub-job (i.e., task) to the node where the requested chunk resides. In other words, instead of moving the chunks of file through the network into a single node to do the decoding, the needed computations (or, tasks) are dispatched directly to the right node for the required data. The rationale is that the overhead of splitting a job into tasks and dispatching them should be significantly smaller than the I/O overhead of moving the potentially large chunks. In addition, the conventional way to read the merged value is in a serial manner, while our proposed approach enable the tasks to process their chunks in parallel.

7.1 Background

7.1.1 Erasure Coding. Erasure coding, together with file replication, are the two major mechanisms to achieve data redundancy. It has been studied by the computer communication community since the 1990's [118, 166], as well as in storage and filesystems [90, 150, 72]. The idea is straightforward: a file is split into k chunks and encoded into $n > k$ chunks, where any k chunks out of these n chunks can reconstruct the original file. We denote $m = n - k$ as the number of redundant chunks (parities). Each chunk is supposed to reside on a distinct disk. Weatherspoon and Kubiatowicz [196] show that for total N machines and M unavailable machines, the availability of a chunk (or replica) A can be calculated as

$$A = \sum_{i=0}^{n-k} \frac{\binom{M}{i}\binom{N-M}{n-i}}{\binom{N}{n}}.$$

Figure 7.1 illustrates what the encoding process looks like. At first glance, the scheme looks similar to file replication, as it allocates additional disks as backups. Nevertheless, the underlying rationale of erasure coding is completely different from file replication for its complex matrix computation.

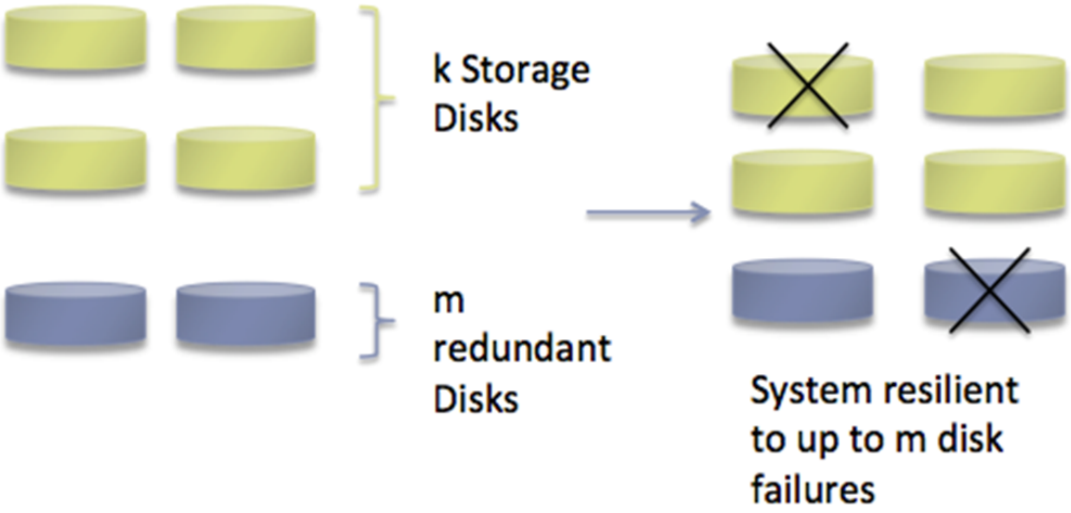


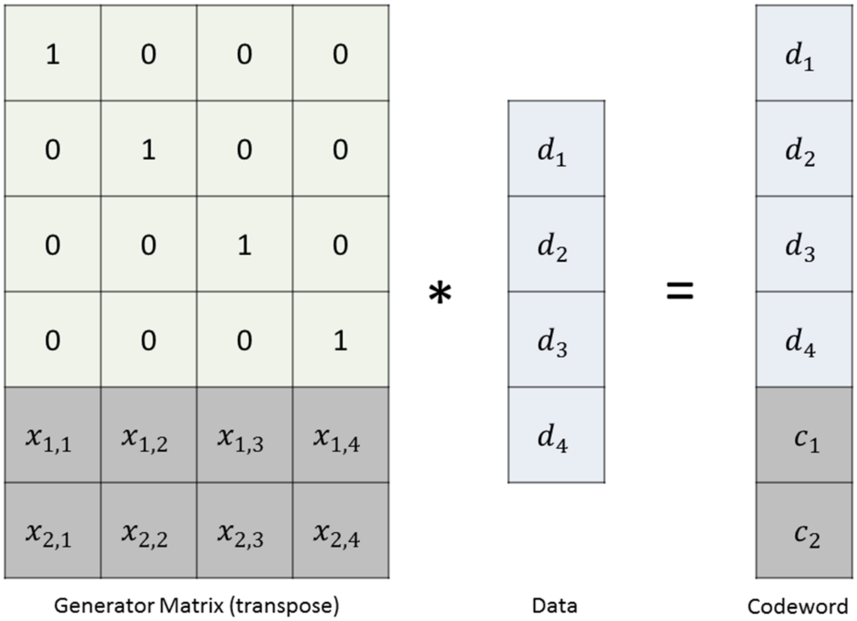
Figure 7.1. Encoding k chunks into $n = k + m$ chunks so that the system is resilient to m failures

As a case in point, one popular erasure code is Reed-Solomon coding [163], which uses a generator matrix built from a Vandermonde matrix to multiply the k data to get the encoded $k + m$ codewords, as shown in Figure 7.2.

Comparing with data replication, erasure coding has 3 important features.

First, erasure coding offers higher space efficiency, defined as $\frac{k}{n}$. This is because redundant parities are smaller than the file itself. Tanenbaum and Steen [183] report that erasure coding outperforms data replication by 40% - 200% in terms of space efficiency.

Second, erasure coding consumes less network bandwidth, because we need not send the entire files but only fractions of them (i.e. parities). This feature is critical



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \end{bmatrix} * \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ c_1 \\ c_2 \end{bmatrix}$$

Figure 7.2. Encoding 4 files into 6 codewords with Reed-Solomon coding

in limited network resources, e.g. geographically dispersed Internet-connected Cloud computing systems built with commodity hardware.

The last and often underrated advantage is security. Rather than copying the intact and non-encrypted file from one node to another, erasure coding chops the file into chunks, then encodes and disperses them to remote nodes. This process is hard to reverse if the encoding matrix is wisely chosen. Moreover, erasure coding-based data redundancy guarantees data security with $(k-1)$ compromised nodes because the minimal number of chunks to restore the original file is k. In contrast, a simple file replication cannot tolerate any compromised nodes; if one node (with the replica) is compromised, the entire file is immediately compromised. Therefore, for applications with sensitive data, erasure coding is the preferred mechanism over replication.

The drawback of erasure coding stems from its computation overhead. It places an extensive burden on the computing chips, so it used to be impractical for production storage systems. This is one of the reasons why prevailing distributed storage systems (e.g. Hadoop distributed file system [178], Google file system [63]) prefer data replication to erasure codes.

7.1.2 GPU Computing.

The graphics processing unit (GPU) was originally designed to rapidly process images for the display. The nature of image manipulations on displays differs from tasks typically performed by the CPU. Most image operations are conducted with single instruction and multiple data (SIMD), where a general-purpose application on a CPU takes multiple instructions and multiple data (MIMD). To meet the requirement of computer graphics, GPU is designed to have many more cores on a single chip than CPU, all of which carry out the same instructions at the same time.

The attempt to leverage GPU's massive number of computing units can be tracked back to the 1970's in [51]. GPUs, however, did not become popular for processing general applications due to its poor programmability until GPU-specific programming languages and frameworks were introduced such as OpenCL [181] and CUDA [137]. These tools greatly eased the development of general applications running on GPUs, thus opened the door to improving applications' performance with GPU acceleration, which are usually named general-purpose computing on graphics processing units (GPGPU). GPGPU gains tremendous research interest because of the huge potential to improve the performance by exploiting the parallelism of GPU's many-core architecture as well as GPU's relatively low power consumption, as shown in [211, 35, 108].

Table 7.1 shows a comparison between two mainstream GPU and CPU devices, which will also be used in the testbeds for evaluation later in this chapter. Although the GPU frequency is only about 50% of CPU, the amount of cores outnumbers CPU by $\frac{336}{6} = 66X$. So the overall computing capacity of GPU is still more than one order of magnitude higher than CPU. This GPU's power consumption should also be noted; only $\frac{0.48}{20.83} = 2.3\%$ of CPU. As energy cost is one of the most challenging problems in large-scale storage systems [168, 86, 219, 218, 207], GPU has the potential to ameliorate it.

Table 7.1. Comparisons of Two Mainstream GPU and CPU

Device	Nvidia GTX460	AMD Phenom
Number of Cores	336	6
Frequency (GHz)	1.56	3.3
Power (W / core)	0.48	20.83

7.2 Analysis

This section presents the analysis of the proposed mechanism for achieving fault tolerance in distributed key-value stores. In particular, we show how the parameters quantitatively affect the system in terms of space utilization and I/O performance. Evaluation on the real system will be presented later.

7.2.1 Assumptions. We assume the multiple paths between the primary copy and the remote nodes have no interference. Therefore, if a full replica is split into 4 chunks and sent to 4 different nodes concurrently, then the transfer speed is roughly reduced to 25% comparing with the full-size replication (assuming the data are already loaded into memory). As mentioned before, the full-size replica is transferred in a serialized manner.

For the encoding and decoding processes, we do not distinguish between the rates between both. This is because literature [40] shows that the difference between the two processes is marginal. It should be noted that this assumption only holds for this analysis section; we will report the performance difference between encoding and decoding procedures.

We also assume the parity code is in the same size of the chunk. In practice, this is the case of most coding algorithms. It also greatly simplifies the analysis in this section.

7.2.2 Parameters. We consider the following parameters when analyzing the abstraction model of Gest. The size of the primary copy is denoted by s. The primary copy is split into k chunks. The number of tolerable failures should be the same as the number of parities in Gest, which is denoted by m. For example, if we request that Gest is resistant to two failed nodes, then m should be set to 2. The coding throughput (both encoding and decoding) is c. The network bandwidth is indicated by b. All these parameters are summarized in Table 7.2.

Table 7.2. Parameters of Gest Environment

Variable	Unit	Meaning
n	Number	Number of nodes
s	Byte	Size of the primary copy
k	Number	Number of chunks
m	Number	Number of parities
c	Byte / second	Coding rate
b	Byte / second	Network bandwidth

7.2.3 Space Utilization. As discussed before, the space utilization (or, storage efficiency) of conventional data replication is

$$E_{rep} = \frac{s}{s \cdot m} = \frac{1}{m}$$

The space utilization for Gest is, however, higher:

$$E_{gest} = \frac{s}{s \cdot \frac{k+m}{k}} = \frac{k}{k+m}$$

Note that, in practice m is a small number (for example, 2) and k is usually set to $n - m$. By doing this, all the nodes are involved in the data redundancy procedure. Therefore the storage efficiency of Gest can be also expressed in an alternative way (assuming all nodes participate in the coding process):

$$E_{gest} = \frac{n-m}{n} = 1 - \frac{m}{n}$$

Again, since m is usually a small integer, and because in a large-scale system n (the total number of nodes) is usually significantly larger than m, the value of E_{gest} is highly close to 1. Also recall that the space efficiency of full-size replication is $\frac{1}{m}$. Consequently, the space efficiency of parity coding is roughly m higher than the conventional replication where m indicates the number of tolerable failures.

A variant of Gest is to keep one primary copy and only apply the coding on replicas. This is for those applications having many read requests. So an intact copy will avoid the frequent decoding procedures. Indeed, such a full-size copy is to trade some space for the improved file-read performance. In this case, the storage efficiency is

$$E^p_{gest} = \frac{s}{s + s \cdot \frac{k+m-1}{k}} = \frac{k}{2k+m-1}$$

Similarly, if we assume $k = n - m$, then

$$E^p_{gest} = \frac{n-m}{2 \cdot (n-m) + m - 1} = 1 - \frac{n-1}{2n-m-1}$$

7.2.4 End-to-End I/O Performance. Both conventional replication and Gest need to load the primary copy from disk to memory, so we do not differentiate them. The difference lies in two parts. First, Gest introduces computational overhead when encoding the data. Second, Gest reduces the network transfer time since smaller chunks are migrated in parallel.

Specifically, the time to transfer the full-size replicas is

$$Time_{rep} = \frac{m \cdot s}{b}$$

Gest, on the other hand, needs to take both the GPU encoding time and the transfer time into account for each encoded chunk:

$$Time_{gest} = \frac{s}{k \cdot c} + \frac{s}{k \cdot b}$$

where the first term represents the GPU coding and the second one is for network transfer.

Therefore, the speedup is

$$Speedup = \frac{Time_{rep}}{Time_{gest}} = \frac{\frac{m \cdot s}{b}}{\frac{s}{k \cdot c} + \frac{s}{k \cdot b}} = \frac{m \cdot k \cdot c}{b + c}$$

If we look at b and c in practice when GPU is leveraged, we usually have $b << c$. For example we will show in later evaluation part that for small m's the GPU coding throughput is higher than 1 GB/s (as opposed to O(100 MB/s) for mainstream hard disks). Consequently, the speedup can be expressed as

$$Speedup \approx m \cdot k$$

In other words, the end-to-end I/O throughput could be improved by mk times when full-size replications is replaced by concurrent parity coding, where m indicates the tolerable failures and k indicates the number of chunks per file. This is also the case when the coding and transfer is pipelined; the $Time_{gest}$ term is essentially degraded to $\frac{s}{k \cdot b}$.

7.3 Gest Distributed Key-Value Storage

Cloud platforms such as Microsoft Azure [122] and Amazon EC2 [8] usually provide a simple yet versatile hashtable-like API (e.g. `set(key,value)`, `value` ← `get(key)`) to its underlying distributed storage. The hashtable API relaxes the conventional POSIX API, simplifies otherwise complicated file operations, enables a unified I/O interface to a large variety of applications, thus gains increasing popularity. The storage subsystems underneath these Cloud platforms are typically implemented as distributed key-value stores, which, despite slight implementation differences, assign the file name as the key and the file content (or, blob) as the value.

The state-of-the-art approach for distributed key-value stores to achieve reliability is replication: several remote replicas are created and updated when the primary copy is touched. Redundant file replicas, however, cause the following issues: (1) space overhead, (2) additional (local) disk I/O, and (3) network bandwidth consumption. As a case in point, if every file has two replicas, the space overhead is roughly 200%, along with tripled disk I/O and network traffic.

This chapter, from a system's perspective, seeks the answer to this burning question: how to *efficiently* achieve key-value store's data reliability with *affordable* overhead from the space, I/O, and network? Rather than proposing a new algorithm or model [204, 203], this work is orthogonal to previous study in that we build a real distributed key-value storage system from the ground up with the following design principles: (1) naive file-level replication need to be replaced by more space-efficient mechanisms, and (2) I/O-intensive operations, if possible, should be transformed to compute-intensive ones without affecting the results. While the first principle is self-explanatory, the second one is because modern computer's computing capacity is orders of magnitude faster than its I/O.

A bird's view of Gest architecture is shown in Figure 7.3. Two services are installed on each Gest node: metadata management and data transfer. Each instance of these two services on a particular node communicates to other peers over the network when requested metadata or files cannot be found on the local node.

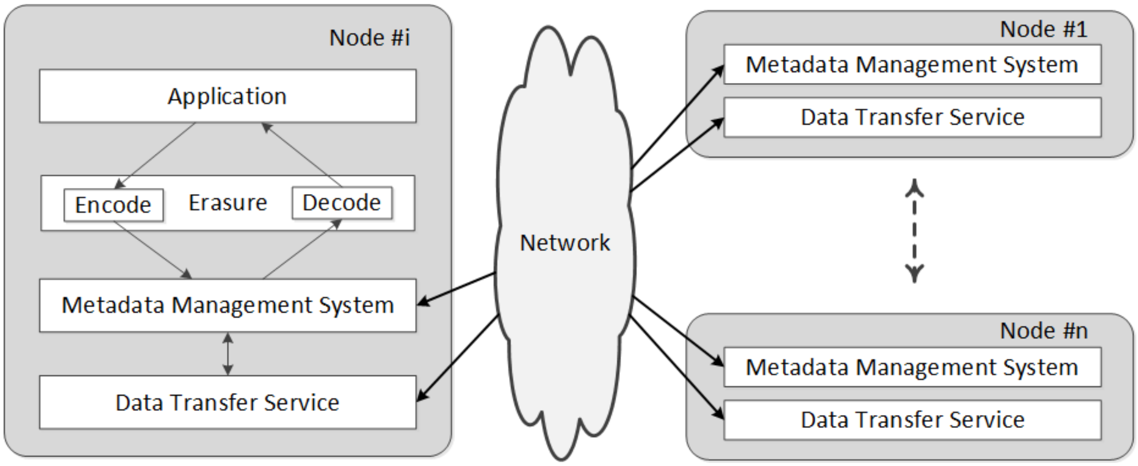


Figure 7.3. Architectural overview of Gest deployed on an n-nodes distributed system

To make matters more concrete, Figure 7.4 illustrates the scenario when writing and reading a file for $k = 4$ and $m = 2$. On the left hand side when the original file (i.e. $orig.file$) is written, the file is chopped into $k = 4$ chunks and encoded into $n = k + m = 6$ chunks. These 6 chunks are then dispersed into 6 different nodes after which their metadata are sent to the metadata hashtable, which is also physically distributed across these 6 nodes. A file read request (on the right hand side) is essentially the reversed procedure of a file write: retrieves the metadata, transfers the chunks, and decodes the file.

7.3.1 Metadata Management. The traditional way of handling metadata for distributed systems is to manipulate them on one or a few nodes. The rational is that metadata contains only high level information (small in size), so a centralized repository usually meets the requirement. Most production distributed storage systems employ centralized metadata management, for instance the Google file system [63] keeps all its metadata on the master node. This design is easy to implement and maintain, yet exposes a performance bottleneck for the workloads generating a large

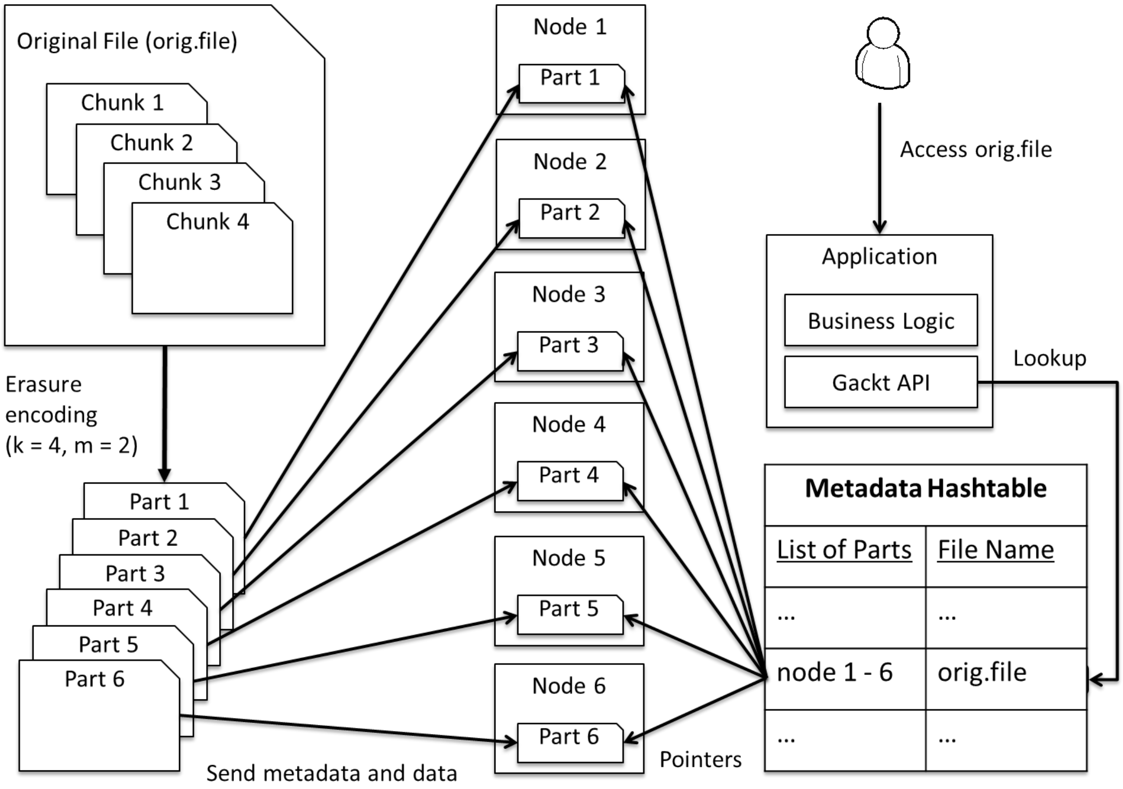


Figure 7.4. An example of file writing and reading on Gest

amount of small files: the metadata rate from a great number of small files can easily saturate the limited number of metadata servers.

In contrast, we implement Gest's metadata management system in a completely distributed fashion. Specifically, all meatdata are dispersed into a distributed hashtable (ZHT [101]). While there are multiple choices of distributed hashtable implementations such as Memcached [56] and Dynamo [44], ZHT has some features that are crucial to the success of serving as a metadata manager.

In Gest, clients have a coherent view of all the files (i.e. metadata) no matter if the file is stored in the local node or a remote node. That is, a client interacts with Gest to inquiry any file on any node. This implies that applications are highly portable across Gest nodes and can run without modifications or recompiling. The metadata and data on the same node, however, are completely decoupled: a file's location has nothing to do with its metadata's location.

Besides the conventional metadata information for regular files, there is a special flag to indicate if this file is being written. Specifically, any client who requests to write a file needs to acquire this flag before opening the file, and will not reset it until the file is closed. The atomic compare-swap operation supported by ZHT guarantees file's consistency for concurrent writes.

7.3.2 Erasure Libraries. Besides daemon services running at the back end, Gest plugs in encoding and decoding modules on the fly. Plank et al. [150] make a thorough review of erasure libraries. In this early version of Gest, we support two built-in libraries Jerasure [148] and Gibraltar [40] as the default CPU and GPU libraries, respectively. Gest is implemented to be flexible enough to support more libraries.

Jerasure is a C/C++ library that supports a wide range of erasure codes: Reed-Solomon coding, Minimal Density RAID-6 coding, Cauchy Reed-Solomon coding, and most generator matrix coding. One of the most popular codes is the Reed-Solomon encoding method, which has been used for the RAID-6 disk array model. This coding can either use Vandermonde or Cauchy matrices to create generator matrices.

Gibraltar is a Reed-Solomon coding library for storage applications. It has been demonstrated to be highly efficient when tested in a prototype RAID system. This library is known to be more flexible than other RAID standards; it is scalable with parity's size of an array. Gibraltar has been created in C using Nvidia's CUDA framework.

7.3.3 Workflows. When an application writes a file, Gest splits the file into k

chunks. Depending on which coding library the user chooses to use, these k chunks are encoded into $n = k + m$ chunks, which are sent to n different nodes by GDT.

At this point the data migration is complete, and we will need to update the metadata information. To do so, ZHT on each of these n nodes is pinged to update the file entries. This procedure of metadata update on the local node is conducted by an in-memory hashmap whose contents are asynchronously persisted to the local disk.

Reading a file is just the reversed procedure of writing. Gest retrieves the metadata from ZHT and uses GDT to transfer (any) k chunks of data to the node where the user makes the request. These k chunks are then decoded by the user-specified library and restored into the original file.

7.3.4 Pipeline. Because the encoded data are buffered, GDT can disperse n encoded chunks onto n different nodes while the file chunks are still being encoded. This pipeline with the two levels of encoding and sending allows for combining the two costs instead of summing them, as described in Figure 7.5.

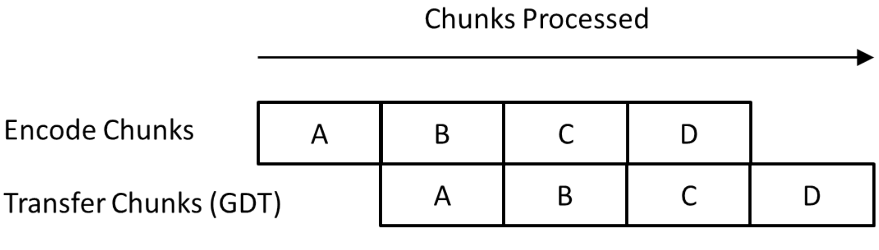


Figure 7.5. Pipelining of encoding and transferring for a write operation in Gest

7.3.5 Client API. Gest provides a completely customizable set of parameters for the applications to tune how Gest behaves. In particular, users can specify which coding library to use, the number of chunks to split the file (i.e. k), the number of parity chunks (i.e. $m = n - k$), the buffer size (default is 1MB), and the like.

7.4 Erasure Coding in FusionFS

Erasure coding is applied to the primary copy as soon as the file is closed. This method avoids the block-level synchronization, and operates before the potential I/O bottleneck of the underlying persistent storage. In FusionFS, erasure coding logic is implemented in the $fusion\_release()$ interface, which is exactly the point right after a file is closed but before it is flushed to the disk. As long as the file is closed (and still in memory), this file is considered "complete", and is ready to be split into n chunks by the erasure libraries. These n chunks are then transferred to n different physical nodes.

7.5 Evaluation

7.5.1 Experiment Design. We compare the conventional file replication to erasure coding mechanisms of different parameter combinations at different scales. The list of candidate mechanisms is summarized in Table 7.3, along with the number of chunks for both the original file and the redundant data.

For file replication, the "chunks of the original data" is the file itself, and the "chunks of redundant data" are plain copies of the original file. We choose 2 replicas for file replication as the baseline, since this is the default setup of most existing systems. Therefore we see `<Replica, 1, 2>` in Table 7.3.

Similarly, Gest with different erasure-coding parameters are listed as `Erasure [1..6]`, along with different file granularity and additional parities. We design experiments at different scales to show how Gest scales. Specifically, every pair of erasure mechanisms represents a different scale: `Erasure[1,2]` for 8-nodes, `Erasure[3,4]` for 16-nodes, and `Erasure[5,6]` for 32-nodes. For example, tuple `<Erasure6, 29, 3>` says that we split the original file into 29 chunks, encode them with 3 additional

Table 7.3. List of Data Redundancy Mechanisms Considered in Gest

Mechanism Name	Chunks of Original File	Chunks of Redundant data
Replica	1	2
Erasure1	3	5
Erasure2	5	3
Erasure3	11	5
Erasure4	13	3
Erasure5	27	5
Erasure6	29	3

parities, and send out the total 32 chunks into 32 nodes.

The numbers of redundant parities (i.e. "chunks of redundant data") for `Erasure[1..6]` are not randomly picked, but in accordance with the following two rules. First, we are only interested in those erasure mechanisms that are more reliable than the replication baseline, because our goal is to build a more space-efficient and faster key-value store without compromised reliability. Therefore in all erasure cases, there are at least 3 redundant parities, which are more than the replica case (i.e. 2). Second, we want to show how different levels of reliability affect the space efficiency and I/O performance. So there is one additional configuration for each scale: the redundant parities are increased from 3 to 5.

7.5.2 Data Reliability and Space Efficiency. Figure 7.6 shows the tolerable failures (i.e. data reliability) and space efficiency for each of the 7 mechanisms listed in Table 7.3. The tolerable failures (histograms) of `Erasure[1..6]` are all more than `Replica`, so is the space efficiency. Thus, when properly configured, erasure codes are better choices than replication in terms of both reliability and efficiency. Before we investigate more about performance, the following conclusions are drawn from Figure 7.6.

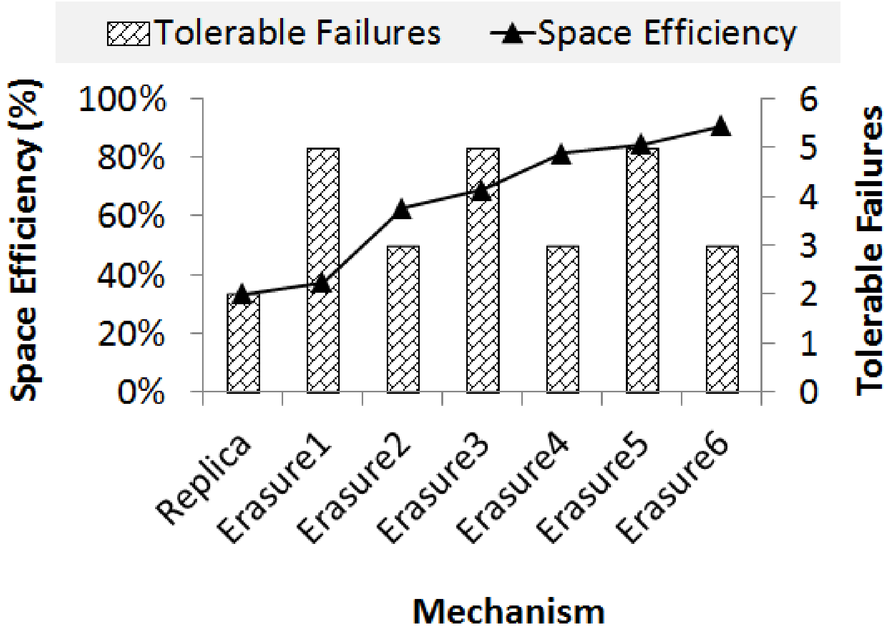


Figure 7.6. Data reliability and space efficiency

First, a larger scale enables higher space efficiency. This is somewhat counter-intuitive, as it is a well-accepted practice to collect data to a small subset of nodes (e.g. collective I/O buffers small and dispersed I/O to reduce the number of small I/Os). Figure 7.6, however, demonstrates that when redundant parity stays the same, space efficiency is monotonic increasing on more nodes. The reason is that with more nodes the redundant parity is in finer granularity and smaller in size. Therefore less space is taken by the redundant data.

Second, for a specific erasure code at a particular scale, reliability and efficiency are negatively correlated. For example, if we increase tolerable failures from 3 to 5, the space efficiency goes from 65% down to 35% (i.e. `Erasure2`→`Erasure1`). This is understandable, as increasing parities take more space.

7.5.3 I/O Performance.

We compare the read and write throughput of all mechanisms listed in Table 7.3 on the HEC cluster at 8-nodes, 16-nodes, and 32-nodes scales. The files to be read and written are 1GB per node, with block size 1MB. The result is reported in Figure 7.7. While most numbers are self-explanatory, some need more explanations in the following.

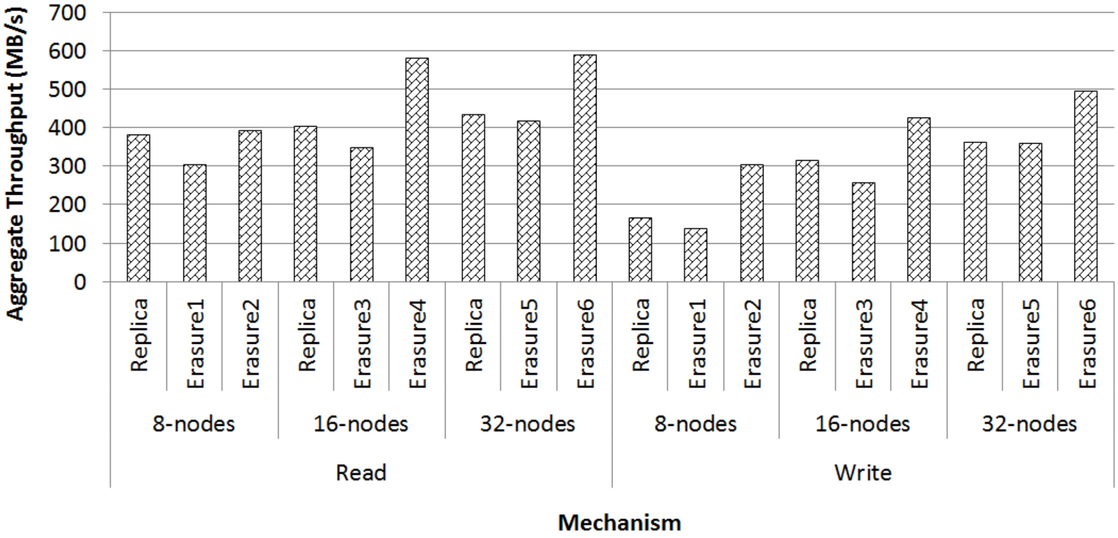


Figure 7.7. Performance on the HEC cluster

One important observation from Figure 7.7 is the promising performance by erasure coding even on CPUs. In many cases (e.g. file read on 16-nodes with Erasure4), Gest delivers higher throughput than the replication counterpart. This is because replication uses more network bandwidth: two extra full-sized replicas introduce roughly a double amount of data to be transferred than erasure coding. We will soon discuss how GPUs further improve Gest performance in Figure 7.10.

Figure 7.7 also shows that, besides the number of nodes, the number of redundant parities greatly impact the I/O performance. Simply increasing the number of nodes does not necessarily imply a higher throughput. For instance, `Erasure2` on 8 nodes delivers higher I/O throughput than `Erasure3` on 16 nodes.

It is worth mentioning that the promising erasure-coding results from HEC should be carefully generalized; we need to highlight that the CPUs of HEC are relatively fast – 8 cores at 2GHz. So we ask: what happens if the CPUs are less powerful, e.g. fewer cores at a lower frequency?

To answer this question, we deploy Gest on Intrepid, where each node only has 4 cores at 850MHz. For a fair comparison, we slightly change the setup of last experiment: the replication mechanism makes the same number of replicas as the additional parities in erasure coding. That is, the reliability is exactly the same for all replication- and erasure-based mechanisms. Moreover, we want to explore the entire parameter space. Due to limited space, we only enumerate all the possible parameter combinations with constraint of 8 total nodes, except for trivial cases of a single original or redundant chunk. That is, we report the performance in the following format `(file chunks: redundant parities)`: (2:6), (3:5), (4:4), (5:3), and (6:2); we are not interested in (1:7) and (7:1), though.

As reported in Figure 7.8, for all the possible parameters of erasure coding, Gest is slower than file replication. As a side note, the throughput is orders of magnitude higher than other testbeds because Intrepid does not have local disk and we run the experiments on RAM disks. Rather than disappointing, this experiment justifies our previous conjecture on the importance of computing capacity to the success of Gest. After all, the result intuitively makes sense; a compute-intensive algorithm needs a powerful CPU. This, in fact, leads to one purpose of this chapter: what if we utilize even faster chips, e.g. GPUs?

Before discussing the performance of GPU-powered Gest at scales, we investigate GPU and CPU coding speed on a single Sirius node. As shown in Figure 7.9,

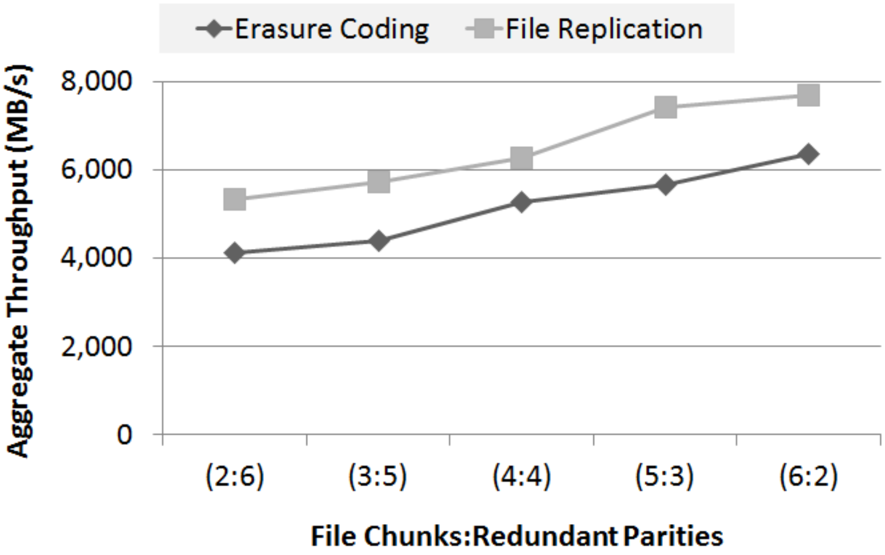


Figure 7.8. Performance on Intrepid

GPU typically processes the erasure coding one order of magnitude faster than CPU on a variety of block sizes (except for encoding 16MB block size: 6X faster). Therefore, we expect to significantly reduce the coding time in Gest by GPU acceleration, which consequently improves the overall end-to-end I/O throughput.

We re-run the 8-nodes experiments listed in Table 7.3 on the Sirius cluster. The number of replicas is set to the same number of redundant parities of erasure coding for a fair comparison of reliability, just like what we did in Figure 7.8. The results are reported in Figure 7.10, where we see for both (5:3) and (3:5) cases, GPU-powered erasure coding delivers higher throughput (read and write combined). Recall that Figure 7.7 shows that CPU erasure-coding outperforms file replication in some scenarios; now Figure 7.10 says that GPU accelerates erasure-coding to outstrip all the replication counterparts.

7.5.4 Erasure Coding in FusionFS. Figure 7.11 shows the throughput of erasure coding- and replication-based data redundancy in FusionFS. Only when $m = 2$, replication (REP) slightly outperforms erasure coding (or IDA, information dispersal

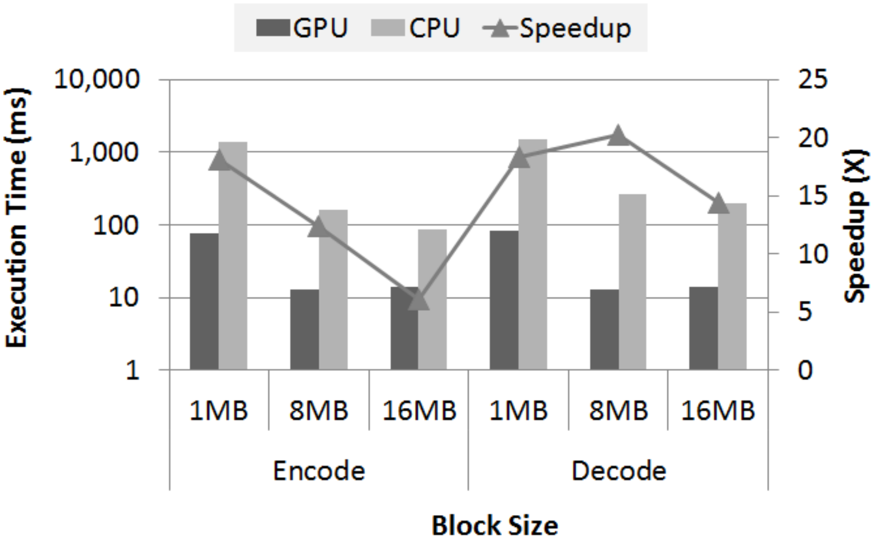


Figure 7.9. Gest coding time on a single Sirius node

algorithm), and the difference is almost negligible. Starting from $m = 3$, IDA clearly shows its advantage over REP, and the speedup increases for the larger m. Particularly, when $m = 6$, i.e. to keep the system's integrity allowing 6 failed nodes, IDA throughput is 1.82 higher than the traditional REP method.

We just showed that in FusionFS installed with a commodity GPU, IDA outperforms REP in terms of both performance and storage utilization (except for the edge case $m = 2$, where IDA and REP are comparable). We believe a high-end GPU would cause a larger gap, and make IDA the top candidate for data redundancy.

7.5.5 Locality-Aware Task Scheduling. We evaluate the performance benefit from locality-aware scheduling by executing MapReduce workloads. The input data is 10 GB extracted from Wikipedia. We do weak-scaling experiments that process 256 MB data per instance. That is, at 128 instances the total data size is 32 GB (i.e., 3.2 copies of the 10 GB input data). The first application is "grep", which searches texts to match the given pattern in the file. The second application is "sort", which performs in-place sort of all the words of a given file. We set $k = 4$ and $m = 2$ in

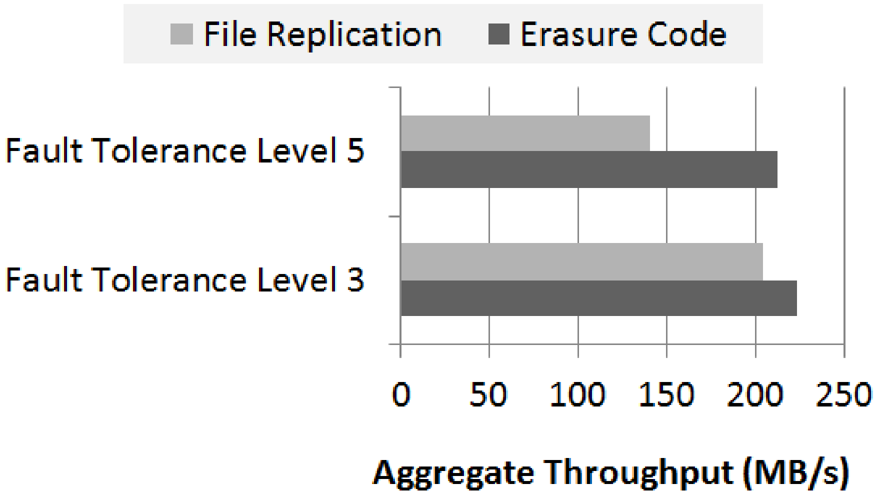


Figure 7.10. Performance on the Sirius cluster

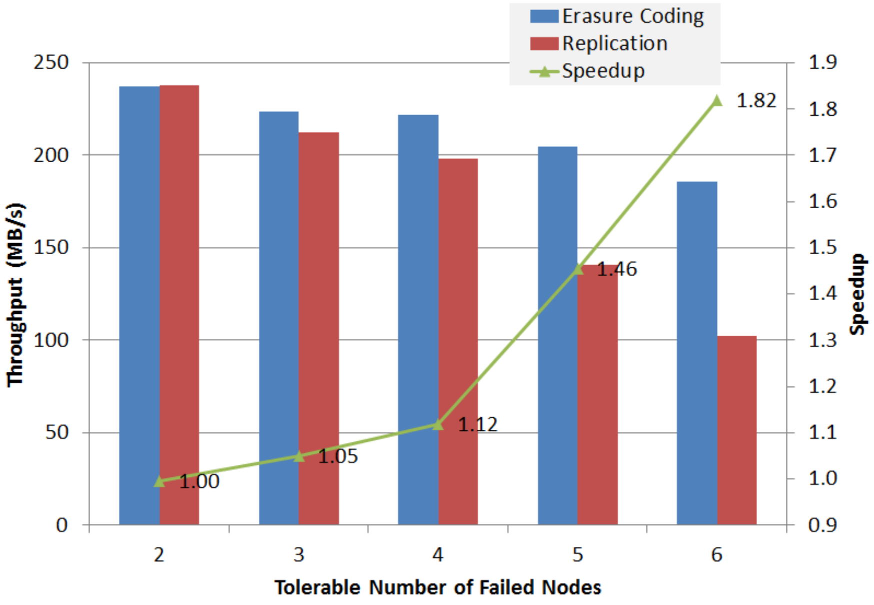


Figure 7.11. Throughput of erasure coding and file replication on FusionFS (block size 1MB)

this experiment; that is, each file is split into 4 equal chunks and encoded with 2 additional parities with GPUs.

Figure 7.12 shows, at different scales from 1 to 128 instances, the speedup and efficiency when 4 tasks are derived from a single job and then work on 4 chunks concurrently. The speedup is measured by considering the scalability, the wall time of the application when chunking files is disabled, and the overhead introduced by the finer granularity of the tasks (sub-jobs). The efficiency is defined as the ratio of the real speedup over the scalability, i.e., the number instances in this case.

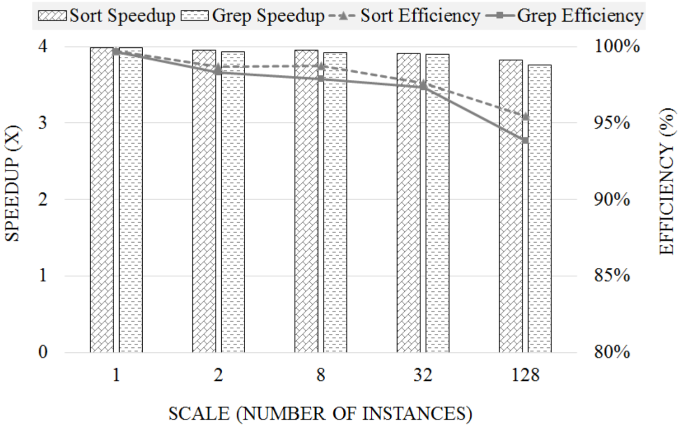


Figure 7.12. Speedup and efficiency of two applications (sort and grep) when data-locality aware scheduling is enabled

We observe that the speedup is slightly decreased from 1 to 128 instances. The reason is that more instances incur higher overhead of spawning a larger number of small tasks. Nevertheless, this overhead is significantly smaller than the I/O gain from the data parallelism. As the efficiency plot shows, the speedup keeps around 95% at 128 instances—we only lose 5% efficiency after scaling up more than two orders of magnitude.

7.6 Summary

This chapter presents Gest, a distributed key-value store whose reliability is based on erasure coding accelerated by GPUs. To the best of our knowledge, Gest is the first distributed key-value store with built-in erasure coding and GPU acceleration. We, from a system's perspective, showcase how to architect and engineer a practical system to solve the long-exiting dilemma between data reliability, space efficiency, and I/O performance. In particular, Gest justifies that key-value stores' reliability can be achieved as the same level as conventional file replication but with superior space efficiency and I/O performance. In a more general sense, Gest demonstrates that a data-intensive problem can be transformed into a compute-intensive one (erasure coding), which is then solved by more powerful computing devices (GPUs). We also demonstrate how to integrate erasure coding into FusionFS. Experiment shows erasure coding is a promising approach for a more space-efficient and faster mechanism than conventional file replication in POSIX filesystems.

CHAPTER 8
PARALLEL SERIALIZATION

Conventional data serialization tools assume that objects to be coded are usually small in size so a single CPU core can encode it in a timely manner. In the era of Big Data, however, object gets increasingly complex and larger, which makes data serialization become a new performance bottleneck. This chapter describes an approach to parallelize data serialization by leveraging multiple cores. Parallelizing data serialization introduces new questions such as how to split the (sub)objects, how to allocate the available cores, and how to minimize its overhead in practice. In this chapter we design a framework for parallelly serializing large objects and analyze the design tradeoffs under different scenarios. To validate the proposed approach, we implemented Parallel Protocol Buffers (PPB)—the parallel version of Google's Protocol Buffers, a widely-used data serialization utility. Experimental results confirm the effectiveness of PPB: multiple cores employed in data serialization achieve highly scalable performance and incur negligible overhead.

8.1 Overview

Serialization is the de facto mechanism for data interchange in distributed systems. In essence, on the client side a data structure (or, an object) is encoded into another format (typically a string) that is transferred to and decoded on the server side. For example, serialization is widely used in remote procedure calls (RPC): the client marshals (i.e., serializes) the procedure parameters and sends the packed message to the server who unmarshals (i.e., deserializes) the message and calls its local procedure with the unpacked parameters; after the local procedure is finished, the server then conducts a similar but reversed process to return the value to the client.

Although only running on a single CPU core, conventional techniques for data serialization are sufficient for most workloads because the objects are usually very small. In the era of Big Data, however, objects of large-scale distributed systems are becoming increasingly larger, which is challenging the viability of our conventional wisdom. As a case in point, at Google our systems experience many RPC messages in the size of hundreds of megabytes. Our serialization tool—Google's Protocol Buffers [153]—was originally not designed for messages of this size; Instead, Protocol Buffers assumes that objects are small enough to be efficiently encoded and decoded with a single CPU core in a serial manner. That is, a gap between the conventional assumption and the real-world situation is increasingly enlarged.

This work explores how to leverage modern computing systems' multi-cores to improve the serialization and deserialization speed of large objects. Rather than proposing new serialization algorithms, we tackle the problem from a system's perspective. Specifically, we propose to leverage multiple CPU cores to split a large object into smaller sub-objects so to be serialized in parallel. While data parallelism is not a new idea in general, it has never been applied to data serialization and poses new problems. For instance, serializing multiple chunks of a large object incurs additional overhead such as metadata maintenance, thread and process synchronization, resource contention. In addition, the granularity (i.e., the number of sub-objects) is a machine-dependent choice: the optimal number of concurrent processes and threads might not align with the available CPU cores.

In order to overcome these challenges and better understand whether the proposed approach could improve the performance of data serialization of large objects, we provide detailed analysis on the system design, for example how to determine the sub-object's granularity for optimal performance and how to ensure that the performance gain is larger than the cost. To demonstrate the effectiveness of our pro-

Table 8.1. A Partial List where Protocol Buffers is Deployed

Organization / Project	Description
Google	All internal projects: MapReduce [43], Google File System [63], Bigtable [34]
Twitter	For efficient and flexible data storage
Apache Camel [12]	Default data interchange format in enterprise integration
SWI-Prolog [182]	Recently added support in logical programming
The R Programming Language	The `RProtoBuf` Package
Protobuf-Embedded-C [174]	For resource constrained applications in embedded systems
FusionFS Filesystem [234]	Metadata management in distributed file systems
ZHT Key-Value Store [101]	Default data interchange format in distributed key value stores

posed approach, we implemented a system prototype called parallel protocol buffers (PPB) by extending a widely-used open-source serialization utility (Google's Protocol Buffers [153]). We have evaluated PPB on a variety of test beds: a conventional Linux server, the Amazon EC2 cloud, and an IBM Blue Gene/P supercomputer. Experimental results confirm that the proposed approach could significantly accelerate the serialization process. In particular, PPB could accelerate the metadata interchange 3.6x faster for an open-source distributed file system (FusionFS [234]).

To summarize, this work makes the following contributions:

- We identify a new performance bottleneck on data serialization in parallel and distributed systems, and propose a data parallelism approach to address it;
- We discuss the design tradeoff and quantitatively analyze the abstracted model under different scenarios;
- We implement a system prototype by extending Google's Protocol Buffers and evaluate it at a variety of test beds.

8.2 Background and Motivation

Although we illustrate the parallelization of data serialization by extending Google's Protocol Buffers [153], the approach is generally applicable to other serialization tools as well. The reason we choose to extend Protocol Buffers is two-fold. First, Protocol Buffers is used as the serialization utility at Google internally, so it is straightforward to work with its intrinsic designs and features. Second, Protocol Buffers is widely used in both production systems and research projects (see Table 8.1 for examples); therefore we hope the community could largely benefit from an extension of this popular tool.

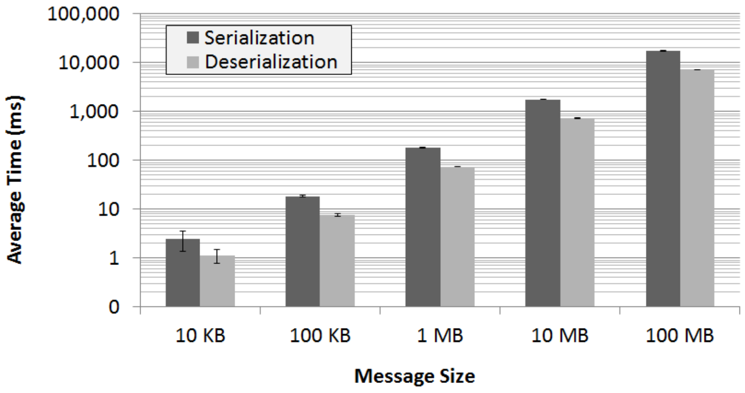


Figure 8.1. Serialization and deserialization with Protocol Buffers

We give a motivating example on the serialization bottleneck at Google. Among many others, one of our servers' tasks is to take the incoming query requests from our enterprise clients. The end-to-end time for completing a RPC is up to a few minutes when the message size is large, and more than 35% of time is spent on data serialization and deserialization. At the same time, the CPU utilization rate is low (i.e., several cores stay idle) even though RPCs require intensive computation to coding large objects. That is, on one hand, the overhead of computing the serialization is significant; on the other hand, many computing resources are not being utilized. Consequently, the coding time is proportional to the size of the message. As shown in Figure 8.1, the (de)serialization time increases at the same rate as the message size on one of our servers. For all message sizes tested in the figure, only one CPU core is busy (more than 98% CPU usage) with the coding job. We only report the performance of message size up to 100 MB in Figure 8.1, as it is on par with the largest objects we observe at Google. Yet, this O(100 MB) is beyond the original design goal of Protocol Buffers, as large messages pose new challenges such as security concerns. In fact, the default maximal message size of Protocol Buffers is 64 MB. For messages between 64 MB and 512 MB, developers need to manually lift the upper limit (for example, the 100 MB case in Figure 8.1). Messages larger than 512 MB are not supported in any release.

8.3 Design and Analysis

This section presents the design and analysis of the proposed parallelism for data (de)serialization. We will first provide a high-level overview of the system architecture, then discuss the design space on how to split and merge the objects, and finally analyze how to choose the number of cores as best practice.

8.3.1 System Architecture. The high-level overview of the system architecture is shown in Figure 8.2. It shows the serialization and deserialization procedures from a sender to a receiver where six CPU cores work concurrently on the six sub-objects constituting the original large object.

It is worth noting, though, that Figure 8.2 only shows one use case of parallel data serialization—the data is interchanged via the network between two nodes. The serialized messages can be used in other scenarios as well such as being persisted to the databases.

8.3.2 Manipulating Objects and Sub-Objects. The first question we need to answer is how to split the given large object. In a homogeneous setting where all cores have the same computing capacity, it is straightforward to evenly split the original

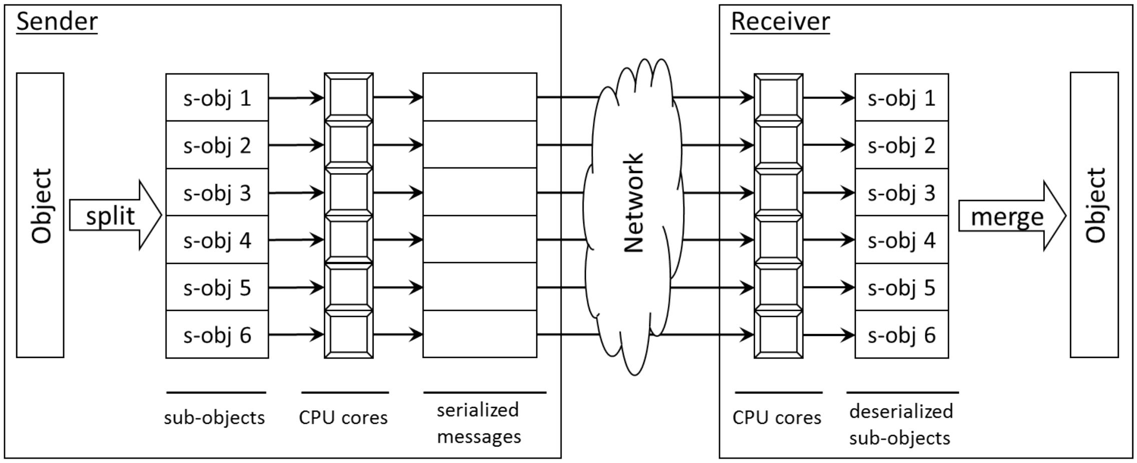


Figure 8.2. System architecture

object. By doing this, the longest execution time for all sub-objects can be minimized. In practice, however, an equidistant splitting is not always possible because some variables might cross the boundary of the sub-objects. Then the question becomes: do we want to split the variable on the boundary?

We argue that a micro splitting at the variable granularity (in addition to the object level) is not worthy to trade for the strict evenness between sub-objects. The benefit of variable-level splitting is highly limited because the variable on the boundary is eventually a primitive data type such as integer, string, float, and so forth. That is, although a nested object could happen to reside on the boundary, we could still split this nested object and in the end it is either a variable or nothing right sitting on the boundary. A primitive variable is usually not larger than a memory page, thus splitting it into two processes brings limited benefit but incurs overhead and complexity. In the real world, here at Google the large messages we deal with comprise a huge number of repeated field (i.e., the data structure to store an array) of primitive data formats, rather than a few and large variables. This fact further justifies the choice not to apply the variable-level splitting.

Since splitting a variable is not a good choice in this context, the best we can do to approximate the evenness between sub-objects is to adjust the sub-objects' boundary to align with the variables'. In the following analysis, however, we assume that all sub-objects are of equal size; this greatly simplifies the analysis and is accurate enough to quantify the proposed approach.

Merging sub-objects at the receiver side is the reversed procedure from system's perspective. Algorithmically, the merging stage is significantly simpler than the splitting one. This is mainly because on the receiver side the number of sub-objects is fixed, as opposed to be decided on the sender side at runtime.

8.3.3 Optimal Number of Cores. This and the following sections quantitatively discuss some good practice in choosing the number of cores for the data parallelism in data (de)serialization. The environment parameters taken into consideration are listed in Table 8.2. The end-to-end wall time of the original serialization and deserialization is denoted by T. When the workload is dispersed to multiple cores, overhead H is introduced by each core such as creating the thread or process, metadata update, and multicasting of messages. Lastly, the number of cores to be leveraged for the data parallelism is denoted by N.

Table 8.2. Environment Variables to Parallelize Data (De)Serialization

Variable	Description
T	Single-core data interchange time
H	Per-core overhead of parallelization
N	Number of cores working in parallel

Since we assume each sub-object is of same size, the time for parallel serialization is $\frac{T}{N}$. Similarly, the overhead by these N cores is $H \cdot N$. We therefore have the end-to-end time for the parallel version of data interchange as a function of N:

$$F(N) = \frac{T}{N} + HN$$

Intuitively, adding more cores reduces the (de)serialization time, but increases the overhead. We are interested in the choice of N to minimize $F(N)$. Suppose $\hat{N}$ is continuous, so if we take the first derivative of $\hat{N}$ ans solve the following equation,

$$\frac{d}{d\hat{N}}(F(\hat{N})) = H - \frac{T}{\hat{N}^2} = 0$$

we have

$$\hat{N} = \sqrt{\frac{T}{H}}$$

In addition, the second-order derivative of $\hat{N}$ is always positive:

$$\frac{d^2}{d\hat{N}^2}(F(\hat{N})) = \frac{T}{\hat{N}^3} > 0$$

Therefore, the optimal number of cores N_{opt} should be

$$\arg\min_N F(N) = \begin{cases} \left\lfloor \sqrt{\frac{T}{H}} \right\rfloor & \text{if } F\left(\left\lfloor \sqrt{\frac{T}{H}} \right\rfloor\right) < F\left(\left\lceil \sqrt{\frac{T}{H}} \right\rceil\right) \\ \\ \left\lceil \sqrt{\frac{T}{H}} \right\rceil & \text{otherwise} \end{cases}$$

Accordingly, the minimal end-to-end time is

$$F_{min}(N) = min\left(F\left(\left\lfloor \sqrt{\frac{T}{H}} \right\rfloor\right),\ F\left(\left\lceil \sqrt{\frac{T}{H}} \right\rceil\right)\right)$$

If $\sqrt{\frac{T}{H}}$ is an integer, a simpler form of the above analysis is

$$N_{opt} = \arg\min_N F(N) = \sqrt{\frac{T}{H}} \tag{8.1}$$

and

$$F_{min}(N) = F\left(\sqrt{\frac{T}{H}}\right) = 2\sqrt{HT} \tag{8.2}$$

Note that in practice H could be significantly smaller than T, making N_{opt} a large number in Equation 8.1.

8.3.4 Performance Gain Guarantee. We just show how to pick the optimal number of cores for the parallelism. In the real world, however, N_{opt} might not be viable; for example the available cores might be (much) fewer than N_{opt}. Therefore, a more realistic, or maybe more interesting, question is how to pick the number of cores to guarantee performance gain. That is, how to choose N to ensure

$$T > F(N) = \frac{T}{N} + HN$$

Again, this is not a trivial problem as the first term of $F(N)$ (i.e., $\frac{T}{N}$) decreases with a larger N while the second one (i.e., $H \cdot N$) increases with a larger N.

The above equation can be transformed to

$$H \cdot N^2 - T \cdot N + T < 0 \tag{8.3}$$

Again, let $\hat{N}$ denote the continuous N in the following analysis. If we solve this equation:

$$H \cdot \hat{N}^2 - T \cdot \hat{N} + T = 0$$

we have

$$\hat{N} = \frac{T \pm \sqrt{T^2 - 4 \cdot H \cdot T}}{2 \cdot H} \tag{8.4}$$

In order to make sure $\hat{N}$ has real values, we need to have this condition:

$$T^2 - 4 \cdot H \cdot T > 0$$

That is, we need to have

$$H < \frac{T}{4} \tag{8.5}$$

In other words, one precondition to apply multiple cores to parallelize data serialization is to have the per-core overhead time smaller than a quarter of the processing time itself.

Note that the parabolic curve in Equation 8.3 is open to the top, therefore if condition in Equation 8.5 is satisfied, then $\hat{N}$ should be set to the value in between the two values in Equation 8.4. That is, the number of cores should fall into the following range to guarantee that the parallelism is beneficial:

$$N \in \left[\ \left\lceil \frac{T - \sqrt{T^2 - 4HT}}{2H} \right\rceil, \ldots, \left\lfloor \frac{T + \sqrt{T^2 - 4HT}}{2H} \right\rfloor \ \right]$$

In practice, the above condition should be easy to satisfy. This is because H is usually a significantly smaller number than T for large messages. Therefore, $\sqrt{T^2 - 4HT}$ could be highly close to T, which consequently sets the lower bound (i.e., $\left\lceil \frac{T-\sqrt{T^2-4HT}}{2H} \right\rceil$) as a small integer. In other words, a small number of cores leveraged in the data parallelism have a high chance to outperform sequential (de)serialization on a single core.

8.4 Implementation

We implement the data parallelism for data serialization and data deserialization on top of Google's Protocol Buffers [153] (C++ version), which is called parallel protocol buffers (PPB). We will discuss some implementation details in the following aspects: user interface, multiprocessing, and multithreading.

8.4.1 User Interface. The long term goal of PPB is to provide the built-in support of data parallelism for large messages. Users will only need to specify the number of concurrent computing cores as an argument during the (de)serialization. Or even better, PPB will probe the current system status (for example, the number of idle cores) and automatically set the concurrency according to the theoretical rules.

At the time of writing, PPB is implemented as a middleware that is loosely coupled from Protocol Buffers' own release. That is, PPB is a wrapper of Protocol Buffers; the (de)serialization requests from applications are redirected to PPB who calls the Protocol Buffers in parallel. The main reason of such an implementation decision is for quick system prototyping and studying the performance improvement quantitatively.

Admittedly, one potential drawback of a middleware is missing some chances of optimizing the code as a whole. For example some performance penalty is expected compared to the build-in support of data parallelism. Nevertheless, as we will demonstrate in the evaluation section, a loosely coupled middleware can significantly improve the performance, making the proposed approach more promising when the built-in implementation is completed.

8.4.2 PPB with Multiprocessing. Our first attempt for parallelizing Protocol Buffers is using message passing interface (MPI). In essence, MPI is a well-developed interface to implement multiprocessing applications. That is, the application is split into multiple processes and executed in parallel.

The overhead of MPI implementation of Protocol Buffers mainly comes from MPI initialization (i.e., `MPI_Init, MPI_Comm_rank, MPI_Comm_size`) and MPI synchronization (i.e., `MPI_Barrier, MPI_Finalize`). Fortunately, modern CPUs mostly comprise O(10) cores; such level of concurrency does not cause significant overhead on the MPI part if the targeted message is as large as O(100 MB). There are other communication cost associated with MPI, such as `MPI_Send, MPI_Recv`; but these are dominated by the network cost if we are talking about big messages.

There is also other overhead coming from the operating system. For instance, every new process needs to be forked with a new independent memory stack; as another example, for any inter-process communication a socket needs to be created. This type of overhead exists by nature, and is hardly eliminated as long as we want to apply some forms of parallelism. Yet, we can try to reduce it; an intuitive option is to use multithreading over multiprocessing.

8.4.3 PPB with Multithreading. Comparing with multiprocessing, multithreading saves some resource on memory stacks as all threads share the same heap. Moreover, no sockets are required since they are all within the same process. This is, thus, particularly an advantage for those workloads that are CPU-bound.

Nevertheless, a parallel data serialization does not only involve CPU computation, but also frequent and concurrent memory accesses. This complicates the implementation choice for multithreading because of the possible contention on shared resources. In fact, concurrent write accesses are not supported in Protocol Buffers [153] for thread safety. Note that multiprocessing does not suffer from this because every process owns its complete memory map isolated from others.

To verify that multithreading does not help PPB, we use OpenMP [142] to parallelize the serialization process on a multi-core server (i.e., Fusion, whose detailed specification will be provided in the evaluation section) and report the results in Figure 8.3. As we expect, multiple threads (in a single process) actually degenerate the overall performance.

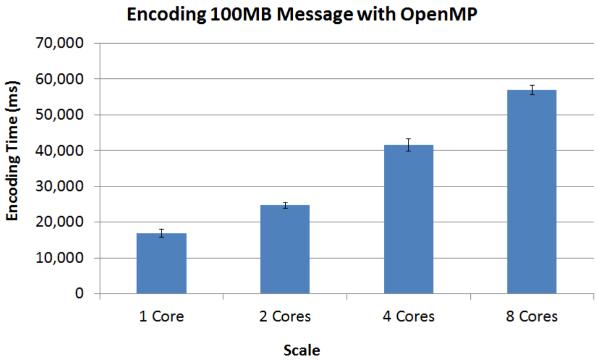


Figure 8.3. Parallel Serialization with OpenMP

Another critical disadvantage of multithreading is the extendibility to multiple nodes. In essence, a multithreading programming paradigm (for example, Pthread, OpenMP) targets only single-node environments. Therefore, if CPU cores are idle in a remote server, the local multithreading program is not able to leverage them. On the other hand, multiprocessing paradigm (for example, MPICH [126], OpenMPI [141]) does not have such a limit of working only on a single node.

After considering all factors we believe multiprocessing is a more appropriate implementation choice than multithreading. In the following discussions, the PPB is a parallel version of Protocol Buffers [153] implemented by MPI.

8.5 Evaluation

8.5.1 Experiment Setup. The test beds where we evaluate PPB are the Fusion Linux server, the Amazon EC2 cloud platform, and an IBM Blue Gene/P supercomputer (i.e., Intrepid [79]). Fusion is a 48-core Linux server with 256 GB RAM and one 2 TB HDD and one 100 GB SSD. For Amazon EC2, the instance types where PPB is deployed are summarized in Table 8.3. Intrepid has 40K nodes in total, each of which is equipped with quad-core 850 MHz PowerPC 450 processors and runs a light-weight Linux ZeptoOS [212] with 2 GB memory.

Table 8.3. Representative Instance Types of Amazon EC2 Cloud Platform

Name	Category	Cores	CPU	Type	Memory	Storage
t2.medium	General Purpose	2	2.5 GHz	Xeon Family	4 GB	EBS only
m3.xlarge	General Purpose	4	2.5 GHz	Xeon E5-2670v2	15 GB	80 GB
m3.2xlarge	General Purpose	8	2.5 GHz	Xeon E5-2670v2	30 GB	160 GB
c3.8xlarge	Compute Optimized	32	2.8 GHz	Xeon E5-2680v2	60 GB	640 GB
r3.8xlarge	Memory Optimized	32	2.5 GHz	Xeon E5-2670v2	244 GB	640 GB
i2.8xlarge	Storage Optimized	32	2.5 GHz	Xeon E5-2670v2	244 GB	6,400 GB

All results are obtained from the PPB middleware implemented with MPI. The MPI library we use is MPICH 3.0.4. The C++ compiler is g++ 4.4.1. The base Protocol Buffers version is 2.6.0.

All experiments are repeated at least five times until results become stable (within 5% margin of error). The reported numbers are the average of all runs. The standard derivations are also plotted along with the averages whenever available.

8.5.2 PPB on the Conventional Server. Figure 8.4 shows the serialization time of parallel protocol buffers for different message sizes on 1 to 32 cores on the Fusion Linux server. For messages larger than 1 MB, the parallelism obviously shows its advantages with excellent scalability. For instance, while coding a 100 MB message takes about 20 seconds with a single core, the same workload only takes 1 second when 32 cores are used. For small messages such as 10 KB and 100 KB, adding more cores do not necessarily improve the performance because the overhead incurred by the parallelism outweighs the savings on the coding.

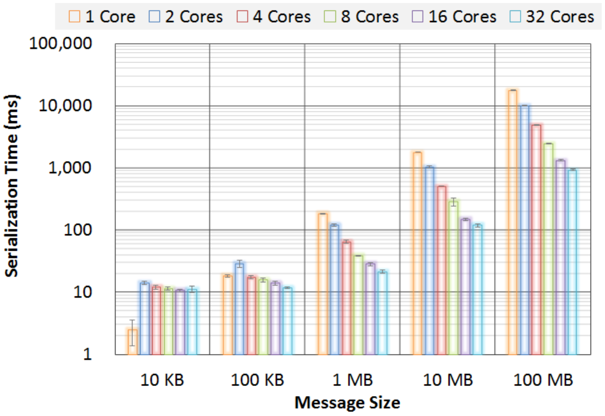


Figure 8.4. Serialization time of PPB on Fusion server

We show the speedup of the above experiment in Figure 8.5. We observe that larger message sizes have closer performance to the theoretical upper bound—the absolutely linear scalability. This is because for larger messages the MPI overhead takes a smaller portion of the overall execution time.

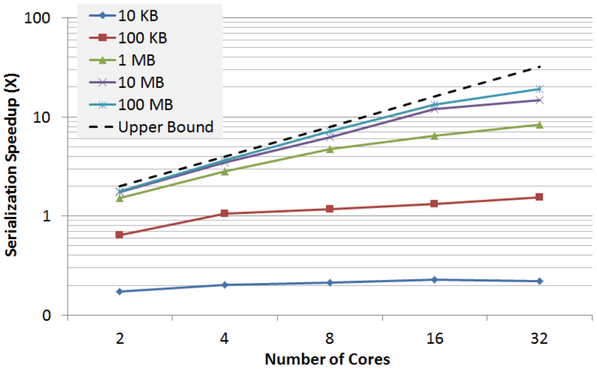


Figure 8.5. Serialization speedup of PPB on Fusion server

Figure 8.6 shows the deserialization time of parallel protocol buffers for different message sizes on 1 to 32 cores. As opposed to the serialization case for small messages, we observe that PPB is as effective as for large messages. The results suggest that the MPI overhead of deserialization is smaller than that of serialization, thus is almost negligible even for small messages.

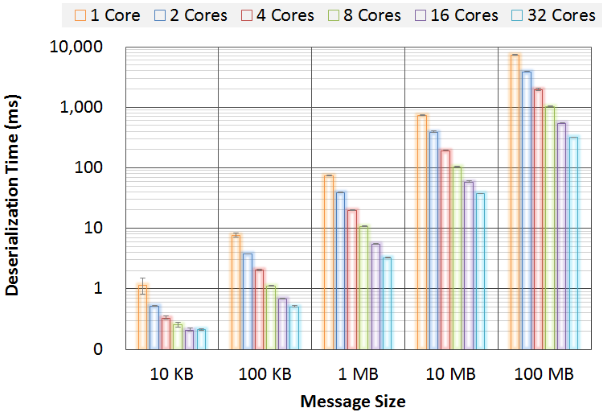


Figure 8.6. Deserialization time of PPB on Fusion server

We show the speedup of the parallel deserialization in Figure 8.7. Similarly to the serialization case, a larger message has a better scalability. In this experiment, a 1 MB seems to be the cut-off size to saturate the 8+ cores as the speedup is almost identical to 1 MB, 10 MB, and 100 MB messages on 8, 16, and 32 cores.

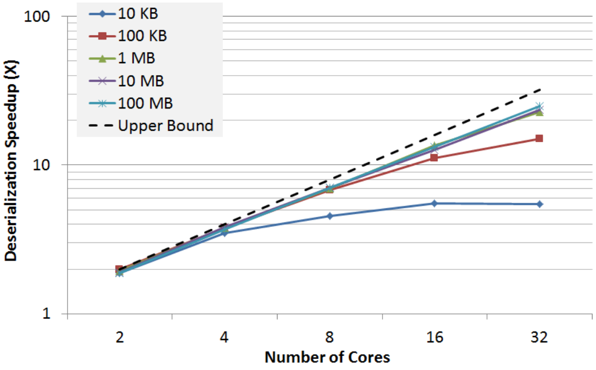


Figure 8.7. Deserialization speedup of PPB on Fusion server

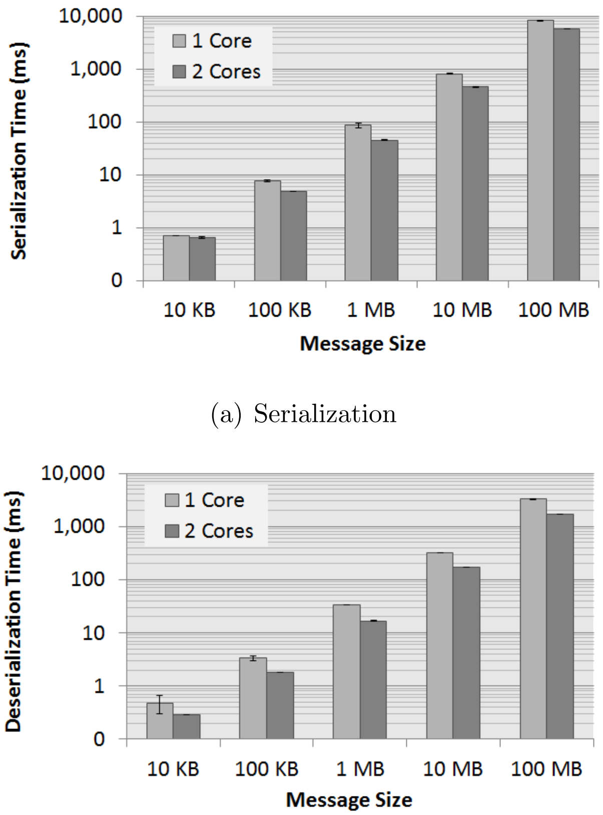
8.5.3 PPB on the Cloud.

In order to understand the viability of parallel serialization in a more general setup particularly for cloud computing, we deploy PPB on a variety of Amazon EC2 instances. In the remainder of this section, we will use cores and vCPUs interchangeably because the latter is to indicate the number of computing units in Amazon EC2. We will not report the speedup due to limited space.

First of all, we report the effect of parallel serialization on a relatively less powerful instance—t2.medium. Figure 8.8(a) and Figure 8.8(b) show the serialization and deserialization time for different sizes of messages, respectively. Recall that a t2.medium instance has 2 vCPUs (i.e., cores), therefore we carried out the experiments on both a serial process (1 Core) and two concurrent processes (2 Cores) scenarios.

From Figure 8.8(a) we learn that a small message such as 10 KB might benefit little from the data parallelism during serialization. Nevertheless, when the message size increases we observe significant saving when serializing the message (i.e., 100 KB to 100 MB). Therefore, when serializing small messages on small EC2 instances, a parallel version could bring limited benefit to the performance.

For deserialization in Figure 8.8(b), however, we observe that both small and large messages could benefit the data parallelism. This result suggests that we apply



(a) Serialization

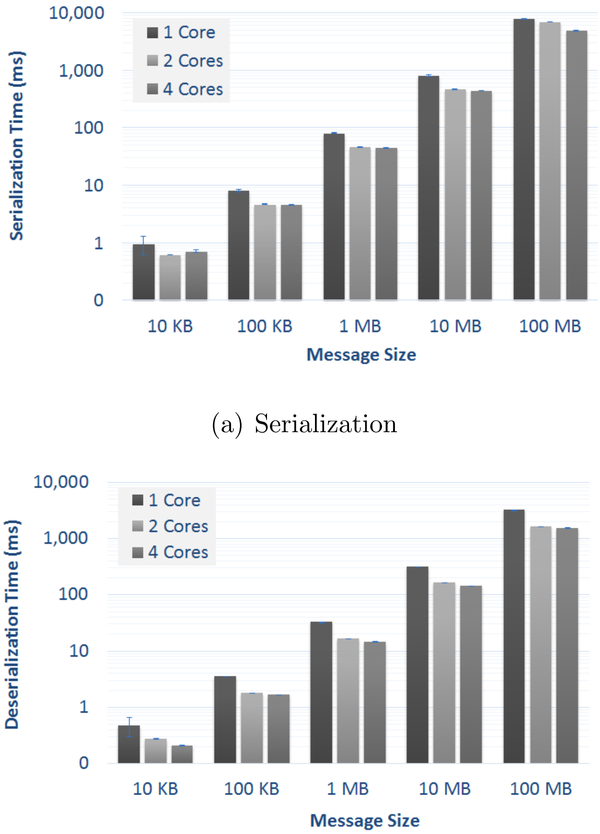
(b) Deserialization

Figure 8.8. PPB on the EC2 t2.medium instance

the parallel version of data deserialization regardless the message size for small EC2 instances.

The evaluation results for a 4-core instance (m3.xlarge) are reported in Figure 8.9(a) and Figure 8.9(b). The execution time is at the same level of t2.medium for 1-core and 2-core.

Figure 8.9(a) shows that for serializing small message (i.e., 10 KB) more cores do not necessarily improve the performance. This is because the overhead for splitting the original message and the multiplied disk write overhead outweigh the gain from the parallelism. Moreover, we observe that even for medium to large messages, 4-core improvement over 2-core is not as significant as that of 2-core over 1-core. We



(a) Serialization

(b) Deserialization

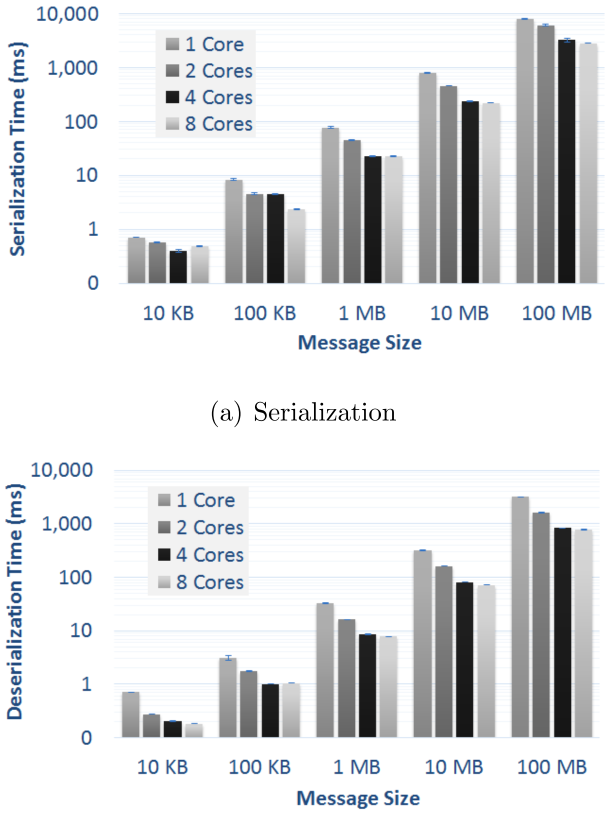
Figure 8.9. PPB on EC2 m3.xlarge instance

believe it can be best explained as that in m3.xlarge virtual machine 4 vCPUs impose significant switching overhead that offsets the data parallelism.

Figure 8.9(b) shows the deserialization performance on m3.xlarge, which has a similar trend as serialization. That is, 4 cores do not bring significant improvement as 2 cores. The only exception is 10 KB where 4 cores almost scale linearly over 2 cores. Yet this is not our major goal as the original serial deserialization is fast enough (i.e., within 1 ms).

For the largest instance in the general purpose category—m3.2xlarge, we rerun the same workload and report the results in Figure 8.10(a) and Figure 8.10(b). We observe that in most cases 8 cores do not help improve much the performance.

In some scenarios 8 cores even degrade the performance, such as serializing 10 KB and deserializing 100 KB. Therefore, although adding cores exploits more data parallelism, over-decomposing the data could cause performance degradation due to the potentially huge divide-conquer overhead.

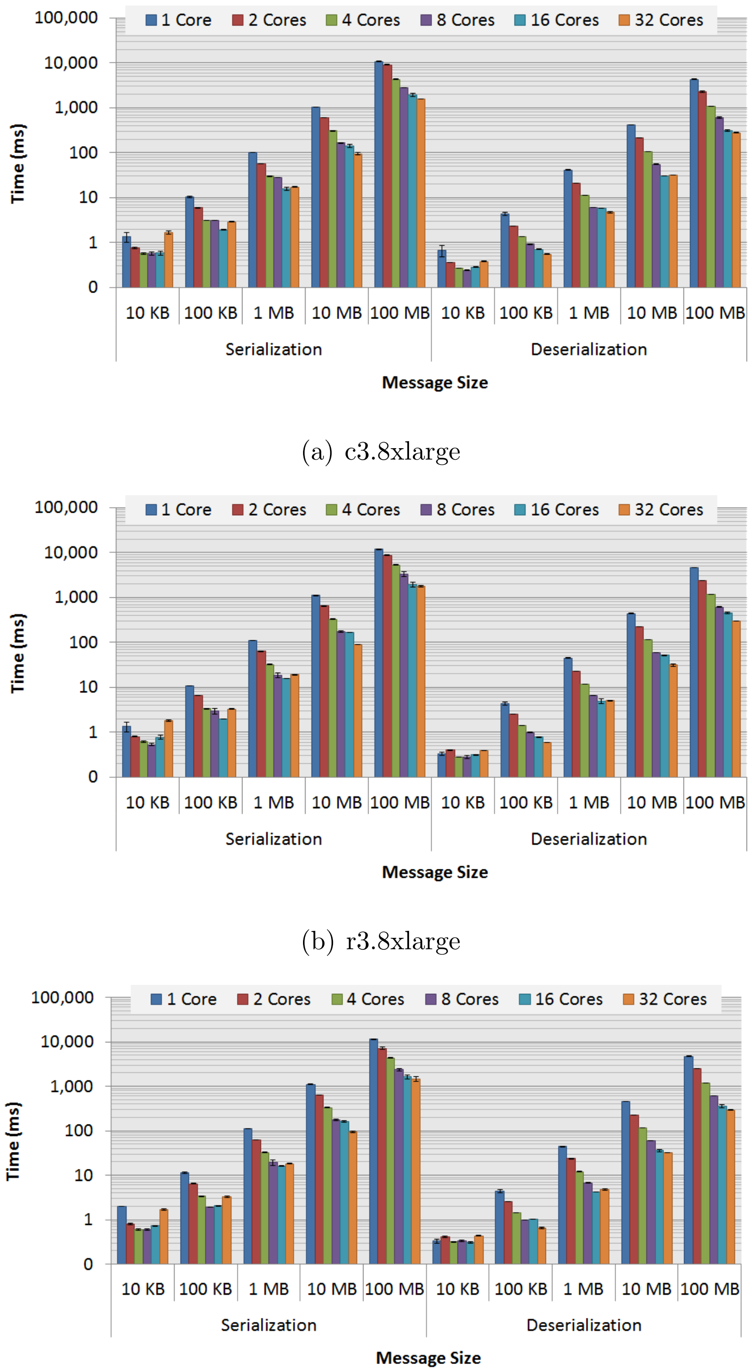


(a) Serialization

(b) Deserialization

Figure 8.10. PPB on EC2 m3.2xlarge instance

Lastly, we report the results of the 32-core instance types from the 3 optimized categories (i.e., compute-optimized, memory-optimized, and storage-optimized) in Figure 8.11. Due to limited space, only the largest instances are considered and compared; that is, 16-cores and smaller cases are not shown here. Again, a similar trend is observed: for small messages the parallelism is not obvious for improved performance; it is the large message where PPB significantly outperforms Protocol Buffers.



(a) c3.8xlarge

(b) r3.8xlarge

(c) i2.8xlarge

Figure 8.11. Large instances of optimized categories

8.5.4 PPB on the Supercomputer. This section demonstrates one PPB use case where it accelerates the metadata performance in a distributed file system designed for world's top supercomputers. The goal of this experiment is two-fold as follows. First, it illustrates that PPB is beneficial to real-world applications in addition to the micro-benchmarks. Second, it showcases the end-to-end performance improvement in a large-scale distributed system—Intrepid [79], one of world's fastest supercomputers when launched.

Specifically, we demonstrate how PPB could accelerate the processing speed for large directories in the FusionFS [234] file system in the following discussion. A large directory comprises many file entries and poses an unprecedented challenge for metadata management [164] in extreme-scale systems. Before discussing PPB performance at this scale, we provide a brief overview of FusionFS.

The Fusion distributed file system (FusionFS [234]) was initially designed to address the I/O bottleneck in the conventional high-performance computing (HPC) systems. The state-of-the-art architecture of HPC systems have all their data stored in the remote storage nodes (for example, GPFS [172]). Therefore, every single I/O has to be transferred via the interconnect between compute and storage nodes. FusionFS, in order to ameliorate the performance bottleneck of the shared network, breaks the accepted practice of compute-storage separation and manipulates all its metadata and data stored right on the compute nodes. As a result, many I/O requests that used to be transferred over the network are now completed by local system calls. Our evaluations on both a real implementation and a simulation model confirm the superiority of FusionFS over the conventional parallel file system; the peak I/O throughput on 16K nodes of an IBM Blue Gene/P supercomputer (i.e., Intrepid [79]) reached 2.5 TB/s surpassing the fastest production storage system (1.4 TB/s) on world's fastest supercomputer (i.e., Titan [187]).

One big challenge in FusionFS is how to support large directories. In theory, all metadata are distributed and balanced on all nodes because internally FusionFS maintains a distributed hash table (DHT) to manage them. Yet, one issue with DHT is that each directory information is stored as a single key-value pair. That is, the pathname is the key while all its entries are in the value of the key. Therefore, if the directory comprises many entries, the value becomes extremely large. Because the metadata is all distributed, these large key-value pairs need to be transferred between multiple nodes. This is exactly where data (de)serialization plays in: whenever the large directory is touched, it needs to be encoded into a serialized format for network transfer and then decoded back for the file system manipulation.

As a case in point, one of workloads we evaluate for FusionFS metadata is to let each of 1,024 nodes create 10,000 empty files in the same shared directory. That is, we expect a single directory to hold roughly 10 million files. The average length of each file name is 10, meaning that each file entry takes 10 bytes in the metadata. This roughly results in a total 100 MB message to be serialized, transferred, and deserialized.

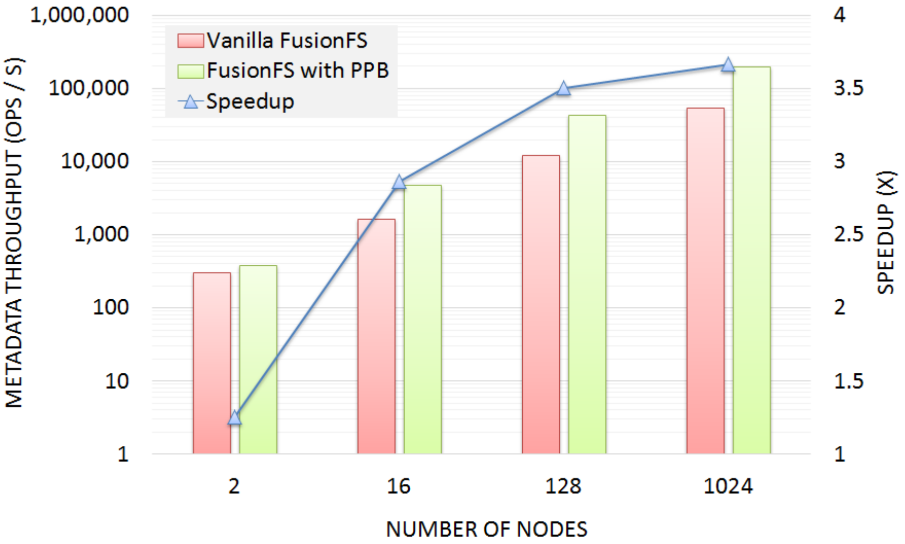


Figure 8.12. FusionFS metadata throughput on Intrepid

Figure 8.12 compares the metadata performance with and without PPB for the workload mentioned above on Intrepid [79]. The red bar measures the performance with single-core Protocol Buffers. Based on our analysis and system configuration, the green bar predicts how the overall performance could be improved by four CPU cores of each Intrepid compute node. For small scale such as 2 nodes, multi-cores do not bring much improvement. Nevertheless, for large scales the gap is significant; in particular, a 3.6X speedup is expected on 1,024 nodes.

8.6 Discussion and Limitation

8.6.1 User Interface. PPB is currently implemented as an MPI wrapper up on Google's Protocol Buffers. Therefore, users would need to make some modifications to application's source code. The change, however, is slight, if not minimal. We name the API names similar to those of Protocol Buffers [153]. For example, the original method `ParseFromCodedStream()`, which is to deserialize the given serialized stream, is now changed to `MPIParseFromCodedStream()` in PPB.

Our long-term goal is to have the parallelism option built in the future release of Protocol Buffers itself. There would be new challenges brought by this design though. One thing worth mentioning is that all the classes and methods provided by Protocol Buffers (the C++ version) are automatically compiled from a "proto"— a user-defined data structure following a C-like syntax. In other words, the parallelism code cannot be directly integrated to the original API; instead, the code change will need to be generated by Protocol Buffers' compiler. We will work closely with Google's Protocol Buffers team to figure out the new design for the parallelism support in the future release.

8.6.2 Dynamic Object Splitting. As mentioned before, current PPB design assumes the serialization occurs in a homogeneous machine. Thus it makes sense to split the large object into sub-objects of the same size. It is a fair question, however, to ask what if the system is heterogeneous.

As a case in point, more and more servers are equipped with high-end GPUs, which comprises a massive number (for example, O(100)) of computing cores. If we want to offload some sub-objects from CPU to GPU, the current design of PPB will need to change. The new design needs to consider the fact that the time for a GPU core to serialize a sub-object of the same size might take quite different time than a CPU core. A straightforward solution would be to split the large object into different sizes of sub-objects. The size would be proportional to the computing capacity (i.e., in GFlops) of CPU and GPU cores. We will also need to consider other factors; for example moving objects between CPU memory and GPU memory also introduces new overhead, which hopefully could be amortized by the performance gain from the massive GPU parallelism.

8.7 Summary

The fact that a significant overhead is incurred when serializing and deserializing large messages in modern parallel and distributed systems calls for a revisit to our conventional wisdom. This work pinpoints the root cause of the performance bottleneck on data serialization and deserialization—serial processing unaware of idle CPU cores, and proposes an unprecedented approach to address the issue. The key idea is to split the data and leverage multiple computing cores to parallelize coding procedures. While data parallelism is not a new idea in general, it has never been applied to data serialization and deserialization, thus posed several new challenges such as divide-conquer strategies and the data-splitting granularity. We explore the design space according to the proposed approach and analyze its abstraction in depth. The system prototype is implemented and deployed on a variety of test beds from a single-node Linux server, to the Amazon EC2 cloud and an IBM Blue Gene/P supercomputer. Experiments confirm the effectiveness of the proposed approach to overcome the new performance bottleneck brought by interchanging large messages in modern distributed systems at scale.

In future we plan to work on the following two main directions of PPB. First, we will deploy PPB on more production systems to test its applicability under various workloads. Second, we will extend the data-parallelism approach to other serialization tools such as Thrift [15].

CHAPTER 9
TOWARDS EXASCALE COMPUTING

This chapter presents the design and implementation of the FusionSim simulator, aimed to simulate FusionFS's performance at the scales beyond today's largest systems—exascale for instance. We will first introduce two building blocks of FusionSim: ROSS [31] and CODES [39], then describe the FusionSim simulator, and finally report the results. Partial results were published in [221].

9.1 ROSS: Discrete Event Simulator

ROSS [31] is a massively parallel discrete-event simulation system developed in Rensselaer Polytechnic Institute. It uses the time warp algorithm and features reverse computation for optimistic simulation. Users can choose to build and run the models in sequential, conservative or optimistic mode. To date, researchers have built many successful large-scale models using ROSS. In [106], ROSS has demonstrated the ability to process billions of events per second by leveraging large-scale HPC systems.

A parallel discrete-event simulation (PDES) system consists of a collection of logical processes (LPs) that are used to model distinct components of the system (for example, a file server in FusionSim). LPs communicate by exchanging time stamped event messages (for example, denoting the arrival or departure of a request message). The goal of PDES is to efficiently process all events in a global timestamp order at the minimum overhead of any processor synchronization.

9.2 CODES: Network Simulation Middleware

CODES [39] is a simulation system built on ROSS. Initially, the goal of CODES is to enable the exploration and co-design of exascale storage systems by providing a detailed, accurate, and highly parallel simulation toolkit for exascale storage systems. As CODES evolves, many modules emerge and have greatly enlarged the original scope. CODES has gradually become a comprehensive platform that supports the modeling and simulation of large-scale complex systems including operating systems, file systems, HPC systems, HPC applications, and data centers.

To date, CODES comprises of the following modules: CODES-net, CODES-workloads, CODES-bg and CODES-lsm. Specifically, CODES-net provides four networking models based on parallel discrete-event simulation: torus network model [105], the dragonfly network model [128], the LogGP model [4] and the simple-net model (a simple N-to-N network).

CODES-net provides unified interfaces that facilitate the use of all underlying networking models. FusionSim leverages the functionality provided by CODES torus network model and thus provides a detailed HPC communication model and simulation. We provide the details of FusionSim in the following section.

9.3 FusionSim: A FusionFS Simulator

With the two enabling techniques, ROSS and CODES, we build FusionSim to simulate FusionFS behavior on the scales beyond today's largest systems. The full software stack is shown in Figure 9.1. Four layers comprise the entire hierarchy of the simulation system (from top downwards): application, file systems, networks, and infrastructure.

When the simulation starts, the system first calls the application layer to generate the workload (i.e., file operations) as the input of the FusionFS file system. Then FusionSim at the file system layer decides where to deal with the file operation depending on its locality. In essence, if this file operation involves network transfer, it is redirected to the network models implemented in the CODES framework at the

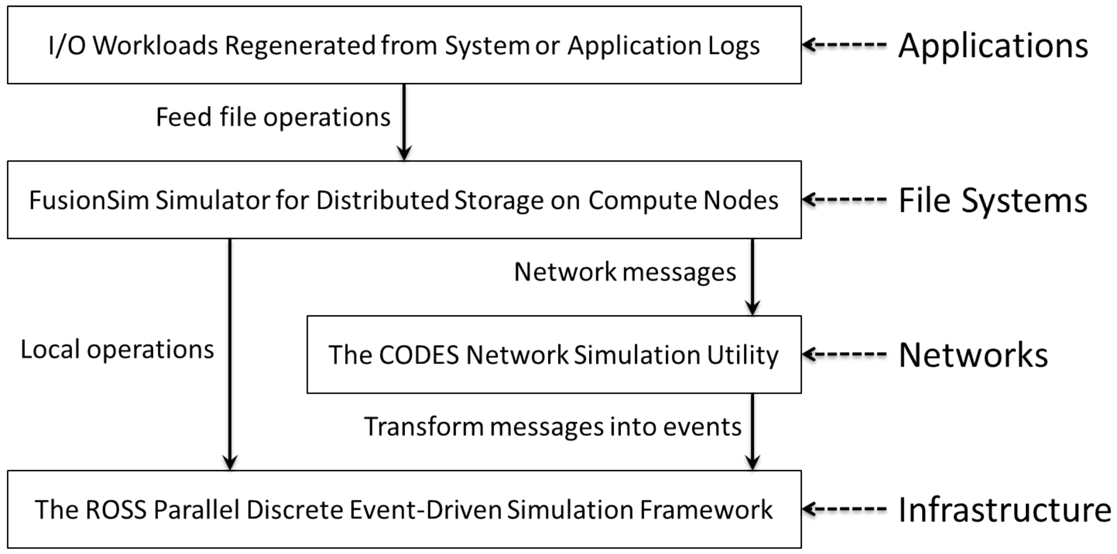

I/O Workloads Regenerated from System or Application Logs
Applications
Feed file operations
FusionSim Simulator for Distributed Storage on Compute Nodes
File Systems
Network messages
Local operations
The CODES Network Simulation Utility
Networks
Transform messages into events
The ROSS Parallel Discrete Event-Driven Simulation Framework
Infrastructure

Figure 9.1. The software stack of FusionSim

network layer. Otherwise, the file operation (i.e., an event) is passed to the ROSS infrastructure. It should be noted that the events at the network layer are eventually passed to the ROSS infrastructure as well.

FusionFS's logic is implemented at the file system layer in Figure 9.1. Basically, each node in FusionFS is abstracted as a logical process in FusionSim. For any file operation (regardless it is related to metadata or data), it triggers a new event in the simulation system. The network topology is implemented at the network layer. For example we can specify the network type for the system to simulate; in this chapter it is set to 3-D torus for the IBM Blue Gene/P supercomputer.

9.4 Evaluation

To show that FusionFS is scalable to even large scales (Intrepid has maximal 40K nodes but requires a special reservation request to conduct experiments at such scales), we build a FusionFS simulator –FusionSim– based on the CODES framework [39]. In particular, we employ the torus network model, and simulate the data transfer between compute nodes as well as the local disk I/O. FusionSim simulates the same workload conducted in Figure 3.10, and is validated by the real FusionFS trace on Intrepid on up to 16K-nodes, as reported in Figure 9.2. The error between the real trace and the simulation result is below 4% at all scales (512-nodes to 16K-nodes), making FusionSim likely accurate in predicting the real system performance at larger scales. In fact, if we take into account the 5% variance from the experiments of FusionFS and FusionSim, the validation error is essentially negligible.

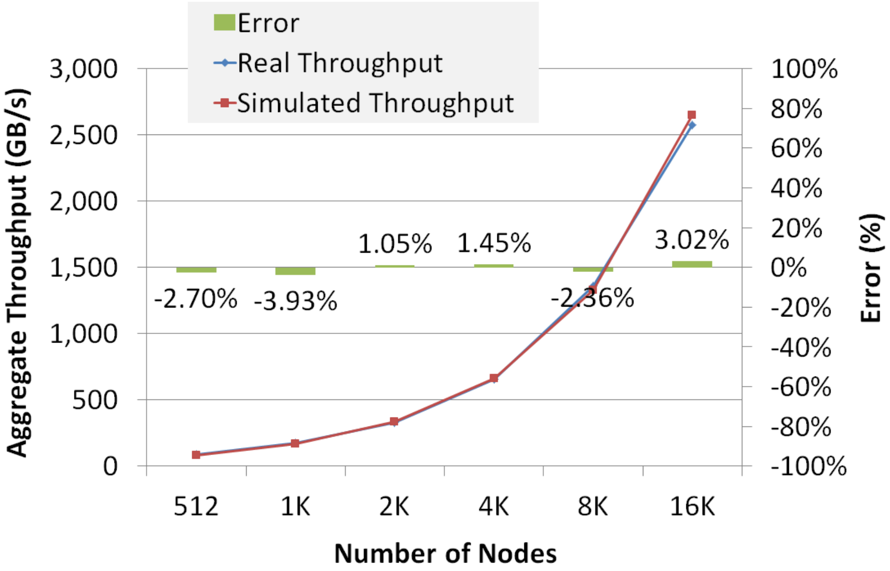


Figure 9.2. Validation of FusionSim on Intrepid

We scale FusionSim with the same workload in Figure 3.10 to 2 million nodes, and report the throughput in Figure 9.3. The ideal throughput is plotted based on the peak throughput 2.64 TB/s achieved at 16K-node scale. FusionFS shows near linear scalability, with more than 95% efficiency at all scales. This can be best explained by its completely distributed metadata operations and the light network traffic involved in data I/O. In particular, FusionFS shows its potential to achieve such an impressive I/O throughput of 329 TB/s on 2 million nodes.

In addition to the aggregate throughput, we feed FusionSim with more complex workloads regenerated from IOR benchmark [80] on GPFS. The workload is

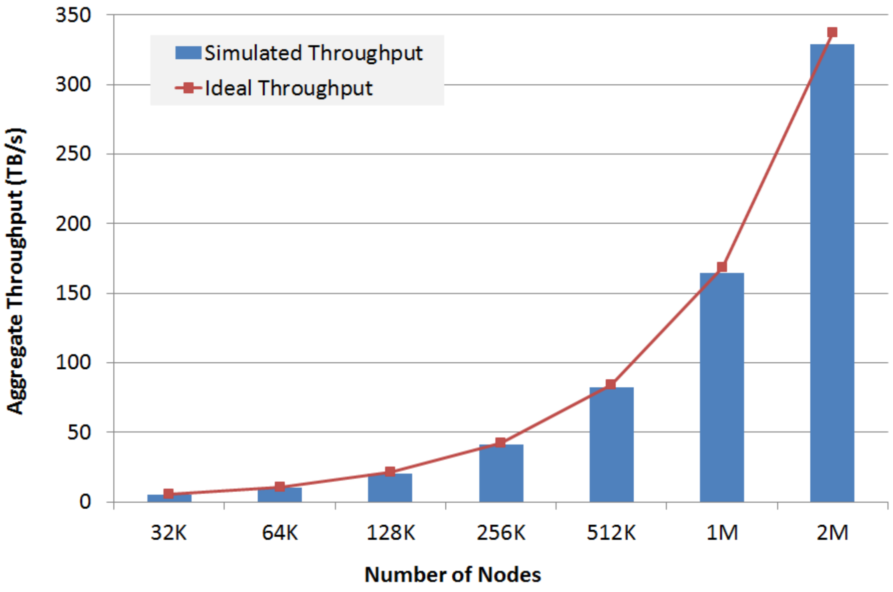


Figure 9.3. Predicted FusionFS throughput by FusionSim

regenerated by the Darshan I/O tool [42] that is statistically equal to the real I/O load. This workload uses MPI collective I/O calls to a shared file in GPFS [172] on a leadership-class supercomputer Intrepid [79]. Each node (i.e., rank) does a sequence of 16 collective writes, closes the file, reopens, does a sequence of 16 collective reads, closes, then exits. All ranks open and close the shared file, and barrier before collective operations. Each rank moves 4 MB per call (64 MB total per rank), which gives us 1 TB total write, and 1 TB total read at 16K scale. Note that this workload does not simply consist of independent I/Os, but involves a lot of collective communication such as data synchronization. We scale this workload from 16K nodes to 1M nodes; so the maximal transferred data is 128 TB on 1M nodes.

Results of IOR [80] workloads are shown in Figure 9.4. For this particular workload, we observe that GPFS is inefficient because the aggregate throughput at the largest scale (i.e., 1M) is only about 37 GB/s that is much smaller than the overall network bandwidth (88 GB/s) between GPFS and compute nodes. On the other hand, FusionFS is predicted to scale linearly and outperforms GPFS at all scales. Note that the speedup only slightly decreases from 16K to 1M nodes.

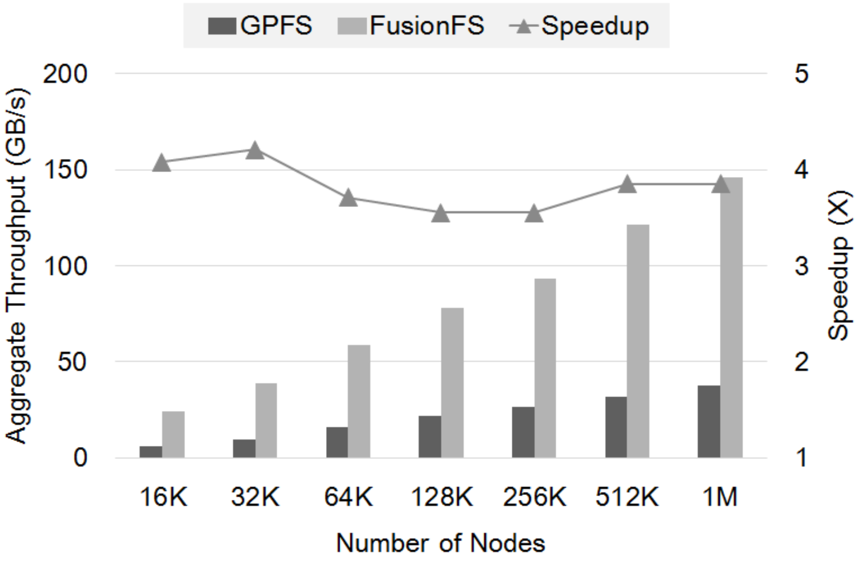


Figure 9.4. Predicted FusionFS performance with IOR, comparing with GPFS

9.5 Summary

This chapter presents the design and implementation of FusionFS's simulator, namely FusionSim, in order to study FusionFS's scalability beyond today's largest machines. FusionSim has two building blocks: the first is a discrete-event simulation infrastructure (ROSS [31]), the second is a simulation middleware for the network (CODES [39]). We validate the accuracy of FusionSim with FusionFS traces on up to 16K nodes, which shows negligible error. We carry out both micro benchmarks and real-world applications with FusionSim on up to 2 million nodes—the scale many experts believe represents exascale computing—with linear scalability.

CHAPTER 10
RELATED WORK

This chapter speaks about related work in distributed storage system from a variety of perspectives.

10.1 Checkpointing

In general, there are two major approaches to checkpointing in HPC systems. The first one is called coordinated checkpointing [27], where all nodes work together to establish a coherent checkpoint. The second one, called Communication Induced Checkpointing (CIC) [7], allows nodes to make independent checkpoints on their local storage. Current HEC systems, e.g. IBM Blue Gene/P supercomputer, adopt the first approach to write states to the parallel filesystem on the network attached storage. Applying CIC is not a viable option in this case, since no local storage is available on the work node of Blue Gene/P.

Some recent works [62, 143] focused on the potentials to substitute traditional hard disks with SSDs on data server to achieve better write bandwidth for checkpointing. Tikotekar et al. [186] developed a simulation framework to evaluate different fault tolerance mechanisms (checkpoint/restart for reactive fault tolerance, and migration for pro-active fault tolerance). The framework uses system failure logs for the simulation with a default behavior based on logs taken from the ASC White at LLNL. A non-blocking checkpointing mode is proposed in [156] to support optimal parallel discrete event simulation. This model allows real concurrency in the execution of state saving and other simulation specific operations (e.g. event list update, event execution), with the aim at removing the cost of recording state information from the parallel application. An incremental checkpointing/restart model is built in [135], which is applied to the HPC environment. The model aims at reducing full checkpointing overhead by performing a set of incremental updates between two consecutive full checkpoints. Some recent research was focused on XOR-based methods, for example, [192] proposed reliable and fast in-memory checkpointing for MPI programs and [37] presented a distributed checkpointing manner using XOR operations. None of these related works explored exascale systems, and none addressed the checkpointing challenges through a different storage architecture (e.g. distributed file systems).

10.2 Parallel and Distributed File Systems

There have been many shared and parallel file systems, such as the Network File System (NFS [50]), General Purpose File System (GPFS [172]), Parallel Virtual File System (PVFS [30]), Lustre[173], and Panasas[134]. These systems assume that storage nodes are significantly fewer than the compute nodes, and compute resources are agnostic of the data locality on the underlying storage system, which results in an unbalanced architecture for data-intensive workloads.

A variety of distributed file systems have been developed such as Google File System (GFS [63]), Hadoop File System (HDFS [178]), Ceph [197], and Sector [68]. However, many of these file systems are tightly coupled with execution frameworks (e.g. MapReduce [43]), which means that scientific applications not using these frameworks must be modified to use these non-POSIX file systems. For those that offer a POSIX interface, they 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 that HPC are anticipating in the coming years.

The idea of distributed metadata can be traced back to xFS [10], even though a central manager is in need to locate a particular file. Recently, FDS [138] 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 component involved. GIGA+ [146] addressed challenges from big directories where millions to billions of small files are created in a single directory. The metadata throughput of GIGA+ significantly outperforms the traditional distributed directory implementations at up to 32-node scales. However, it is not clear if this design would suffice for extreme scales, e.g. 1K nodes and beyond.

Co-location of compute and storage resources has attracted a lot of research interests. For instance, Salus [194] proposes to co-locate the storage to data nodes in data centers. Other examples include Rhea [64], which prevents removing the data used by the computation, and Nectar [70], 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.

10.3 Filesystem Caching

To the best of our knowledge, HyCache is the first user-level POSIX-compliant hybrid caching for distributed file systems. Some of our previous work [158, 161] proposed data caching to accelerate applications by modifying the applications and/or their workflow, rather than the at the filesystem level. Other existing work requires modifying OS kernel, or lacks of a systematic caching mechanism for manipulating files across multiple storage devices, or does not support the POSIX interface. Any of the these concerns would limit the system's applicability to end users. We will give a brief review of previous studies on hybrid storage systems.

Some recent work reported the performance comparison between SSD and HDD in more perspectives ([99, 165]). Hystor [36] aims to optimize of the hybrid storage of SSDs and HDDs. However it requires to modify the kernel which might cause some issues. A more general multi-tiering scheme was proposed in [69] which helps decide the needed numbers of SSD/HDDs and manage the data shift between SSDs and HDDs by adding a 'pseudo device driver', again, in the kernel. iTransformer [214] considers the SSD as a traditional transient cache in which case data needs to be written to the spinning hard disk at some point once the data is modified in the SSD. iBridge [215] leverages SSD to serve request fragments and bridge the performance gap between serving fragments and serving large sub-requests. HPDA [116] offers a mechanism to plug SSDs into RAID in order to improve the reliability of the disk array. SSD was also proposed to be integrated to the RAM level which makes SSD as the primary holder of virtual memory [16]. NVMalloc [191] provides a library to explicitly allow users to allocate virtual memory on SSD. Also for extending virtual memory with Storage Class Memory (SCM), SCMFS [201] concentrates more on the management of a single SCM device. FAST [87] proposed a caching system to pre-fetch data in order to quicken the application launch. In [206] SSD is considered as a read-only buffer and migrate those random-writes to HDD.

A thorough review of classical caching algorithms on large scale data-intensive applications is recently reported in [53]. HyCache+ is different from the classical cooperative caching [151] in that HyCache+ assumes persistent underlying storage and manipulates data at the file level. As an example of distributed caching for distributed file systems, Blue Whale Cooperative Caching (BWCC) [176] is a read-only caching system for cluster file systems. In contrast, HyCache+ is a POSIX-compliant I/O storage middleware that transparently interacts with the underlying parallel file systems. Even though the focus of this chapter lies on the 2-layer hierarchy of a local cache (e.g. SSD) and a remote parallel file system (e.g. GPFS [172]), the approach presented in HyCache+ is applicable to multi-tier caching architecture as well. Multi-level caching gains much research interest, especially in the emerging age of cloud computing where the hierarchy of (distributed) storage is being redefined with more layers. For example Hint-K [200] caching is proposed to keep track of the last K steps across all the cache levels, which generalizes the conventional LRU-K algorithm concerned only on the single level information.

There are extensive studies on leveraging data locality for effective caching. Block Locality Caching (BLC) [120] captures the backup and always uses the latest locality information to achieve better performance for data deduplication systems. The File Access corRelation Mining and Evaluation Reference model (FARMER) [202] optimizes the large scale file system by correlating access patterns and semantic attributes. In contrast, HyCache+ achieves data locality with a unique mix of two principles: (1) write is always local, and (2) read locality depends on the novel 2LS mechanism which schedules jobs in a deterministic manner followed by a local heuristic replacement policy.

While HyCache+ presents a pure software solution for distributed cache, some orthogonal work focuses on improving caching from the hardware perspective. In [104], a hardware design is proposed with low overhead to support effective shared caches in multicore processors. For shared last-level caches, COOP [213] is proposed to only use one bit per cache line for re-reference prediction and optimize both locality and utilization. The REDCAP project [66] aims to logically enlarge the disk cache by using a small portion of main memory, so that the read time could be reduced. For Solid-State Drive (SSD), a new algorithm called lazy adaptive replacement cache [77] is proposed to improve the cache hit and prolong the SSD lifetime.

Power-efficient caching has drawn a lot of research interests. It is worth mentioning that HyCache+ aims to better meet the need of high I/O performance for

HPC systems, and power consumption is not the major consideration at this point. Nevertheless, it should be noted that power consumption is indeed one of the toughest challenges to be overcome in future systems. One of the earliest work [239] tried to minimize the energy consumption by predicting the access mode and allowing cache accesses to switch between the prediction and the access modes. Recently, a new caching algorithm [209] was proposed to save up to 27% energy and reduce the memory temperature up to 5.45°C with negligible performance degradation. EEVFS [115] provides energy efficiency at the file system level with an energy-aware data layout and the prediction on disk idleness.

While HyCache+ is architected for large scale HPC systems, caching has been extensively studied in different subjects and fields. In cloud storage, Update-batched Delayed Synchronization (UDS) [103] reduces the synchronization cost by buffering the frequent and short updates from the client and synchronizing with the underlying infrastructure in a batch fashion. For continuous data (e.g. online video), a new algorithm called Least Waiting Probability (LWP) [205] is proposed to optimize the newly defined metric called user waiting rate. In geoinformatics, the method proposed in [98] considers both global and local temporal-spatial changes to achieve high cache hit rate and short response time.

The job scheduler proposed in this work takes a greedy strategy to achieve the optimal solution for the HyCache+ architecture. A more general, and more difficult, scheduling problem could be solved in a similar heuristic approach [155, 185]. For an even more general combinatorial optimization problem in a network, both precise and bound-proved low-degree polynomial approximation algorithms were reported in [26, 25]. Some incremental approaches [229, 109, 228] were proposed to efficiently retain the strong connectivity of a network and solve the satisfiability problem with constraints.

In future, we plan to better predict the I/O behavior by employing some machine learning techniques such as incremental algorithms [229, 109, 228], as well as more advanced data-aware scheduling mechanisms such as [238, 193].

10.4 Filesystem Compression

While the storage system could be better deigned to handle more data, an orthogonal approach is to address the I/O bottleneck by squeezing the data with compression techniques. One example where data compression gets particularly popular is checkpointing, an extremely expensive I/O operation in HPC systems. In [55], it showed that data compression had the potential to significantly reduce the checkpointing file sizes. If multiple applications run concurrently, a data-aware compression scheme [82] was proposed to improve the overall checkpointing efficiency. Recent study [22] shows that combining failure detection and proactive checkpointing could improve 30% efficiency compared to classical periodical checkpointing. Thus data compression has the potential to be combined with failure detection and proactive checkpointing to further improve the system efficiency. As another example, data compression was also used in reducing the MPI trace size, as shown in [139]. A small MPI trace enables an efficient replay and analysis of the communication patterns in large-scale machines.

It should be noted that a compression method does not necessarily need to restore the absolutely original data. In general, compression algorithms could be categorized into to two groups: lossy algorithms and lossless algorithms. A lossy algorithm might lose some (normally a small) percentage of accuracy, while a lossless one has to ensure the 100% accuracy. In scientific computing, studies [96, 95] show that lossy compression could be acceptable, or even quite effective, under certain circumstances. In fact, lossy compression is also popular in other fields, e.g. the most widely compatible lossy audio and video format MPEG-1 [125]. This section presents virtual chunks mostly by going through a delta-compression example based on XOR, which is a lossless compression. It does not imply that virtual chunks cannot be used in a lossy compression. Virtual chunk is not a specific compression algorithm, but a system mechanism that is applicable to any splittable compression, not matter if it is lossy or lossless.

Some frameworks are proposed as middleware to allow applications call high-level I/O libraries for data compression and decompression, e.g. [20, 171, 83]. None of these techniques take consideration of the overhead involved in decompression by assuming the chunk allocated to each node would be requested as an entirety. In contrast, virtual chunks provide a mechanism to apply flexible compression and decompression.

There is previous work to study the file system support for data compression. Integrating compression to log-structured file systems was proposed decades ago [23], which suggested a hardware compression chip to accelerate the compressing and decompressing. Later, XDFS [113] described the systematic design and implementation for supporting data compression in file systems with BerkeleyDB [140]. MRAMFS [49] was a prototype file system to support data compression to leverage the limited space of non-volatile RAM. In contrast, virtual trunks represent a general technique applicable to existing algorithms and systems.

Data deduplication is a general inter-chunk compression technique that only stores a single copy of the duplicate chunks (or blocks). For example, LBFS [133] was a networked file system that exploited the similarities between files (or versions of files) so that chunks of files could be retrieved in the client's cache rather than transferring from the server. CZIP [144] was a compression scheme on content-based naming, that eliminated redundant chunks and compressed the remaining (i.e. unique) chunks by applying existing compression algorithms. Recently, the metadata for the deduplication (i.e. file recipe) was also slated for compression to further save the storage space [119]. While deduplication focuses on inter-chunk compressing, virtual chunk focuses on the I/O improvement within the chunk.

Index has been introduced to data compression to improve the compressing and query speed e.g. [94, 65]. The advantage of indexing is highly dependent on the chunk size: large chunks are preferred to achieve high compression ratios in order to amortize the indexing overhead. Large chunks, however, would cause potential decompression overhead as explained earlier in this chapter. Virtual chunk overcomes the large-chunk issue by logically splitting the large chunks with fine-grained partitions while still keeping the physical coherence.

10.5 Filesystem Provenance

As distributed systems become more ubiquitous and complex, there is a growing emphasis on the need for tracking provenance metadata along with file system metadata. A thorough review is presented in [129]. Many Grid systems like Chimera [57] and the Provenance-Aware Service Oriented Architecture (PA-SOA) [154] provide provenance tracking mechanisms for various applications. However these systems are very domain specific and do not capture provenance at the filesystem level. The Distributed Provenance Aware Storage System (DPASS) tracks the provenance of files in a distributed file system by intercepting filesystem operations and sending this information via a netlink socket to user level daemon that collects provenance in a database server [145]. The provenance is however, collected in a centralized fashion, which is a poor design choice for distributed file systems meant for extreme scales. Similarly in efficient retrieval of files, provenance is collected centrally [131].

PASS describes global naming, indexing, and querying in the context of sensor data [132], which is a challenging problem also from system's perspective [100]. PA-NFS [130] enhances NFS to record provenance in local area networks but does not consider distributed naming explicitly. SPADE [61] addresses the issue by using storage identifiers for provenance vertices that are unique to a host and requiring distributed provenance queries to disambiguate vertices by referring to them by the host on which the vertex was generated as well as the identifier local to that host.

Several storage systems have been considered for storing provenance. ExSPAN [237] extends traditional relational models for storing and querying provenance metadata. SPADE supports both graph and relational database storage and querying. PASS has explored the use of clouds [132]. Provbase uses Hbase to store and query scientific workflow provenance [1]. Further compressing provenance [237], indexing [114] and optimization techniques [74] have also been considered. However, none of these systems have been tested for exascale architectures. To give adequate merit to the previous designs we have integrated FusionFS with SPADE as well as considered FusionFS' internal storage system for storing audited provenance.

10.6 Data Serialization

10.6.1 Data Interchange. Many serialization frameworks are developed to support transporting data over distributed systems. XML [52] represents a set of rules to encoding documents or text-based files. Another format, namely JSON [88], is treated as a lightweight alternative to XML in web services and mobile devices as well. While XML and JSON are the most widely used data serialization format for text-based files, binary format is also gaining its popularity. A binary version of JSON is available called BSON [21]. Two other famous binary data serialization frameworks are Google's Protocol Buffers [153] and Apache Thrift [15]. Both frameworks are designed to support lightweight and fast data serialization and deserialization, which could substantially improve the data communication in distributed systems. The key difference between Thrift and Protocol Buffers is that the former has the built-in support for RPC.

Many other serialization utilities are available at the present. Avro [11] is used by Hadoop for serialization. Internally, it uses JSON [88] to represent data types and protocols and improves the performance of the Java-based framework. Etch [13] supports more flexible data models(for example, trees), but it is slower and generates larger files. BERT [19] supports data format compatible with Erlang's binary serialization format. Message Pack [121] allows both binary data and non UTF-8 encoded strings. Hessian [75] is a binary web service protocol that is 2X faster than the Java serialization with significantly smaller compressed data size. ICE [78] is a middleware platform that supports object-oriented RPC and data exchange. CBOR [32] is designed to support extremely small message size.

None of the aforementioned systems, however, support data parallelism. Thus they suffer the low efficiency problem when multiple CPU cores are available particularly when the data is large in size. PPB, on the other hand, takes advantage of the idle cores and leverage them for parallelizing the compute-intensive process of data serialization

10.6.2 Parallel Data Processing. Many frameworks are recently developed for parallel data processing. MapReduce [43] is a programming paradigm and framework that allows users to process terabytes of data over massive-scale architecture in a matter of seconds. Apache Hadoop [14] is one of the most popular open-source implementations of MapReduce framework. Apache Spark [210] is an execution engine which supports more types of workload than Hadoop and MapReduce.

Several parallel programming models and paradigms have been existing for decades. Message Passing Interface (MPI) a standard for messages exchange between processes. It greatly reduces the burden from developers who used to consider detailed protocols in multiprocessing programs and tries to optimize the performance in many scenarios. The major implementation includes MPICH [126] and Open MPI [141]. OpenMP [142] is a set of compiler directives and runtime library routines that enable the parallelization of code's execution over shared memory multi-processor computers. It supports different platforms and processor architectures, programming languages, and operating systems. Posix Threads (Pthread) is defined as a set of C programming types and function calls. It provides standardized programming interface to create and manipulate threads, which allow developers to take full advantage of the capabilities of threads. Microsoft's Parallel Patterns Library (PPL) [152] gives an imperative programming model that introduces parallelism to applications and improves scalability.

Numerous efforts have been devoted to utilizing or improving data parallelism in cluster and cloud computing environment. Jeon et al. [84, 85] proposed adaptive parallelization and prediction approaches for search engine query. Lee et al. [97] presented how to reduce data migration cost and improve I/O performance by incorporating parallel data compression on the client side. Klasky et al. [91] proposed a parallel data-streaming approach with multi-threads to migrate terabytes of scientific data across distributed supercomputer centers. Some work [195, 208, 33, 2] proposed data-parallel architectures and systems for large-scale distributed computing. In [6, 190], authors exploited the data parallelism in a program by dynamically executing sets of serialization codes concurrently.

Unfortunately, little study exists on data parallelism for data serialization, mainly because large messages are usually not the dominating cost by convention. This work for the first time identifies that large message is a challenging problem from our observations on real-world applications at Google. We hope our PPB experience could provide the community insights for designing the next-generation data serialization tools.

CHAPTER 11
CONCLUSION AND FUTUREWORK

This dissertation presents a thorough review of conventional system architectures for large-scale big data applications particularly in scientific computing. It is one of the pioneer works that quantitatively predicts the performance of various architectures with extensive simulations. Simulation results illustrate that the conventional architecture would not be viable for the future scales due to the costly network traffic between the compute and storage cliques. In the meantime, simulations show that a new architecture —co-location of compute and storage— would greatly mitigate the I/O pressure and demonstrate a linear scalability towards millions of nodes.

Inspired by the promising results from the simulation, we designed and implemented a system prototype, namely FusionFS, to justify the idea of co-locating compute and storage. FusionFS had two major design goals that are critical to achieve high performance of scientific applications at extreme scales: distributed metadata and independent local file writes. The distributed metadata is realized by a distributed hash table that showed a strong scalability on up to 8K nodes. We designed several data movement protocols to make file writes independent for every client; With these protocols we observed more than 2.5 TB/s aggregate throughput on 16K nodes. FusionFS also showed its performance advantage on several real-world scientific applications.

With the success of the FusionFS system, we further explored other directions with FusionFS as the underlying platform. Our first study is extending FusionFS to be a transparent caching layer (HyCache and HyCache+) between local compute node and remote parallel file systems. In other words, instead of being a standalone file system functioning only on compute nodes, FusionFS is incorporated into the entire ecosystem for scientific computing, from local compute client, to local file system, to a distributed caching layer on compute resources, to the remotely connected storage nodes. The goal of this incorporation is two-fold. First, this design allows users to deal with large file sizes that cannot fit in FusionFS. Note that FusionFS is often deployed on the local memory-class storage and often has capacity limitations. The second goal is to make those non-I/O-intensive applications almost unaffected by the architectural change of FusionFS; After all, FusionFS is crafted for data-intensive applications and has little impact to compute-intensive applications.

Then we switch to study whether we could leverage FusionFS to track and query file systems' provenance. The conventional technique of provenance is maintain and query a centralized relational database, which has many shortcomings such as poor scalability and heavy overhead during concurrent queries. We proposed to leverage distributed hash tables to make both the storage and query of provenance completely distributed. We implemented a system prototype on top of the FusionFS system and carried out extensive evaluations showing that a distributed manner is a promising approach to achieve both efficient provenance storage and lightweight provenance query at the same time.

We also explored the possibility of incorporating more intelligent compression at the file system layer. The conventional way to compress and decompress large files at the file system layer is straightforward: the file is compressed before being written to the disk; it is then decompressed after reading from disk to memory before returning the handle to the users. This approach is completely agnostic about the underlying data layout and important metrics such as compression ratio and decompression overhead — all these are delegated to the compression algorithms with little being done from system's perspective. To this end, we proposed a smarter way to allow the system arbitrarily choose a virtual chunk to compress and decompress. By doing this, we could leverage the data layout and I/O pattern in order to achieve both high compression ratio and low decompression overhead. Experiments with several scientific datasets confirmed the effectiveness of the proposed approach.

We then studied how to make the conventional data reliability more efficient. The state-of-the-art technique for data reliability is replication — every primary copy of any data has several secondary replicas that would become a primary one when the original primary copy is failed or compromised. The problem with this conventional technique lies on the inefficient space usage and consequently the I/O cost. We borrowed the idea of RAID in disks (i.e., erasure coding) that only adds a small spatial overhead but preserves a high data reliability. Yet, the high computation cost to apply erasure codes directly to the data is computationally prohibitive. To this end, we proposed to leverage GPUs to lower the computational overhead in the erasure coding process. The combined effect of GPU computing and erasure coding achieves the best of both worlds: low computational overhead and high spacial utilization.

During the development of FusionFS, we encountered a challenge of serializing big data sets because the building block (i.e., Google's protocol buffer) is not designed for parallel processing. Collaborated with Google, we proposed and experimented a parallel implementation of protocol buffer. The system prototype demonstrated that MPI could greatly improve the performance of protocol buffer for large data sets.

Because today's largest system only comprises tens of thousands of nodes, we need to simulate FusionFS at larger scale to justify its effectiveness at future exascales. We built a simulator, namely FusionSim, based on a parallel discrete event-driven simulation framework. FusionSim was validated with the real-time traces of the FusionFS system at up to 16K nodes. We then scaled FusionSim to up to 2 million nodes — the scale that many believe represents exascale. Both micro-benchmarks and real-world applications on FusionSim suggest that the architecture of co-locating compute and storage would scale almost linearly towards exascales.

In future, we would productize the system prototype and conduct a more thorough comparison with other popular systems on a variety of test beds: in-house clusters, public/private clouds, grids, and supercomputers. A preliminary comparison between FusionFS and Ceph [197] was recently published in [230]. We will also open source the system prototype and allow the community to explore other research directions with this infrastructure.

BIBLIOGRAPHY

[1] J. Abraham, P. Brazier, A. Chebotko, J. Navarro, and A. Piazza, "Distributed storage and querying techniques for a semantic web of scientific workflow provenance," in *Services Computing (SCC), 2010 IEEE International Conference on*. IEEE, 2010, pp. 178–185.

[2] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka, "Large-scale data visualization using parallel data streaming," *Computer Graphics and Applications, IEEE*, vol. 21, no. 4, Jul 2001.

[3] S. Albers, N. Garg, and S. Leonardi, "Minimizing stall time in single and parallel disk systems," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation," in *Proceedings of Annual ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[5] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable i/o forwarding framework for high-performance computing systems," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, 2009.

[6] M. D. Allen, S. Sridharan, and G. S. Sohi, "Serialization sets: A dynamic dependence-based parallel execution model," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09, 2009.

[7] L. Alvisi, E. Elnozahy, S. S. Rao, S. A. Husain, and A. Del Mel, "An analysis of communication induced checkpointing," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, 1999.

[8] Amazon EC2, "http://aws.amazon.com/ec2," Accessed March 6, 2015.

[9] C. Ambühl and B. Weber, "Parallel prefetching and caching is hard," in *STACS*, 2004.

[10] 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.

[11] Apache Avro, "http://avro.apache.org/," Accessed December 13, 2014.

[12] Apache Camel, "http://camel.apache.org/," Accessed December 7, 2014.

[13] Apache Etch, "https://etch.apache.org/," Accessed December 13, 2014.

[14] Apache Hadoop, "http://hadoop.apache.org/," Accessed September 5, 2014.

[15] Apache Thrift, "https://thrift.apache.org/," Accessed December 8, 2014.

[16] A. Badam and V. S. Pai, "SSDAlloc: hybrid SSD/RAM memory management made easy," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011.

[17] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," *J. ACM*, vol. 9, no. 1, Jan. 1962.

[18] A. Benoit, H. Larcheveque, and P. Renaud-Goud, "Optimal algorithms and approximation algorithms for replica placement with distance constraints in tree networks," in *IEEE 26th International Symposium on Parallel Distributed Processing (IPDPS)*, 2012.

[19] BERT, "http://bert-rpc.org/," Accessed December 13, 2014.

[20] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt, "Integrating online compression to accelerate large-scale data analytics applications," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

[21] Binary JSON, "http://bsonspec.org/," Accessed December 13, 2014.

[22] M. S. Bouguerra, A. Gainaru, L. B. Gomez, F. Cappello, S. Matsuoka, and N. Maruyam, "Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing," in *Parallel Distributed Processing, IEEE International Symposium on*, 2013.

[23] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "On-line data compression in a log-structured file system," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[24] bzip2, "http://www.bzip2.org," Accessed September 5, 2014.

[25] G. Calinescu, S. Kapoor, K. Qiao, and J. Shin, "Stochastic strategic routing reduces attack effects," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, 2011.

[26] G. Calinescu and K. Qiao, "Asymmetric topology control: Exact solutions and fast approximations," in *IEEE International Conference on Computer Communications (INFOCOM '12)*, 2012.

[27] G. Cao and M. Singhal, "On coordinated checkpointing in distributed systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 12, dec 1998.

[28] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," *SIGMETRICS Perform. Eval. Rev.*, vol. 23, no. 1, May 1995.

[29] 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 Symposium on Parallel and Distributed Processing*, 2009.

[30] 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.

[31] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: A high-performance, low memory, modular time warp system," in *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, ser. PADS '00, 2000.

[32] CBOR, "http://cbor.io/," Accessed December 13, 2014.

[33] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, Aug. 2008.

[34] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, Jun. 2008.

[35] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, 2008.

[36] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proceedings of the International Conference on Supercomputing*, 2011.

[37] G.-M. Chiu and J.-F. Chiu, "A new diskless checkpointing approach for multiple processor failures," *IEEE Trans. Dependable Secur. Comput.*, vol. 8, no. 4, Jul. 2011.

[38] E. Cohen and M. Strauss, "Maintaining time-decaying stream aggregates," in *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2003.

[39] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothersy, and R. B. Ross, "CODES: Enabling co-design of multi-layer exascale storage architectures," in *Workshop on Emerging Supercomputing Technologies*, 2011.

[40] M. L. Curry, A. Skjellum, H. Lee Ward, and R. Brightwell, "Gibraltar: A reed-solomon coding library for storage applications on programmable graphics processors," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 18, 2011.

[41] J. Daly, "A model for predicting the optimum checkpoint interval for restart dumps," in *Proceedings of the 2003 International Conference on Computational Science*, 2003.

[42] Darshan, "http://www.mcs.anl.gov/research/projects/darshan/," Accessed April 26, 2015.

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

[44] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

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

[46] Y. Ding, H. Tan, W. Luo, and L. Ni, "Exploring the use of diverse replicas for big location tracking data," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, June 2014.

[47] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 2, Jun. 2011.

[48] 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.

[49] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt, "Mramfs: A compressing file system for non-volatile ram," in *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, 2004.

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

[51] J. N. England, "A system for interactive modeling of physical curved surface objects," 1978, pp. 336–340.

[52] Extensible Markup Language (XML), "http://www.w3.org/xml/," Accessed December 13, 2014.

[53] R. Fares, B. Romoser, Z. Zong, M. Nijim, and X. Qin, "Performance evaluation of traditional caching policies on a large system with petabytes of data," in *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*, 2012.

[54] FermiCloud, "http://fclweb.fnal.gov/," Accessed March 6, 2015.

[55] K. B. Ferreira, R. Riesen, D. Arnold, D. Ibtesham, and R. Brightwell, "The viability of using compression to decrease message log sizes," in *Proceedings of International Conference on Parallel Processing Workshops*, 2013.

[56] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, Aug. 2004.

[57] I. T. Foster, J.-S. Vckler, M. Wilde, and Y. Zhao, "The virtual data grid: A new model and architecture for data-intensive collaboration," in *CIDR'03*, 2003.

[58] P. A. Freeman, D. L. Crawford, S. Kim, and J. L. Munoz, "Cyberinfrastructure for science and engineering: Promises and challenges," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 682–691, 2005.

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

[60] GCRM, "http://kiwi.atmos.colostate.edu/gcrm/," Accessed September 5, 2014.

[61] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Proceedings of the 13th International Middleware Conference*, 2012.

[62] S. Gerhold, N. Kaemmer, A. Weggerle, C. Himpel, and P. Schulthess, "Pageserver: High-performance ssd-based checkpointing of transactional distributed memory," in *Computer Engineering and Applications (ICCEA), 2010 Second International Conference on*, vol. 1, march 2010.

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

[64] 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.

[65] Z. Gong, S. Lakshminarasimhan, J. Jenkins, H. Kolla, S. Ethier, J. Chen, R. Ross, S. Klasky, and N. F. Samatova, "Multi-level layout optimization for efficient spatio-temporal queries on isabela-compressed data," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[66] P. Gonzalez-Ferez, J. Piernas, and T. Cortes, "The ram enhanced disk cache project (redcap)," in *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007.

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

[68] 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.

[69] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, 2011.

[70] 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.

[71] Gzip, "http://www.gnu.org/software/gzip/gzip.html," Accessed September 5, 2014.

[72] J. L. Hafner, V. Deenadhayalan, K. K. Rao, and J. A. Tomlin, "Matrix methods for lost data reconstruction in erasure codes," 2005.

[73] HDF5, "http://www.hdfgroup.org/HDF5/doc/index.html," Accessed September 5, 2014.

[74] T. Heinis and G. Alonso, "Efficient lineage tracking for scientific workflows," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1007–1018.

[75] Hessian, "http://hessian.caucho.com/," Accessed December 13, 2014.

[76] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft Research, 2009.

[77] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng, "Improving flash-based disk cache with lazy adaptive replacement," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, 2013.

[78] ICE, "http://doc.zeroc.com/display/ice34/home," Accessed December 13, 2014.

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

[80] IOR Benchmark, "https://asc.llnl.gov/sequoia/benchmarks/IOR_summary_v1.0.pdf," Accessed November 5, 2014.

[81] IOZone, "http://www.iozone.org," 2014.

[82] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "McrEngine: A scalable checkpointing system using data-aware aggregation and compression," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[83] J. Jenkins, E. R. Schendel, S. Lakshminarasimhan, D. A. Boyuka, II, T. Rogers, S. Ethier, R. Ross, S. Klasky, and N. F. Samatova, "Byte-precision level of detail processing for variable precision analytics," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[84] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Adaptive parallelism for web search," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013.

[85] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Predictive parallelization: Taming tail latencies in web search," in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, ser. SIGIR '14, 2014.

[86] X. Jiang, M. Alghamdi, J. Zhang, M. Assaf, X. Ruan, T. Muzaffar, and X. Qin, "Thermal modeling and analysis of storage systems," in *Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International*, 2012.

[87] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "FAST: Quick application launch on solid-state drives," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, 2011.

[88] JSON, "http://www.json.org/," Accessed December 8, 2014.

[89] J. Katcher, "Postmark: A new file system benchmark," in *Network Appliance, Inc.*, vol. 3022, 1997.

[90] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[91] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney, "Grid-based parallel data streaming implemented for the gyrokinetic toroidal code," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03, 2003.

[92] T. Kobus, M. Kokocinski, and P. T. Wojciechowski, “Hybrid replication: State-machine-based and deferred-update replication schemes combined,” in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ser. ICDCS ’13, 2013.

[93] Kodiak, “https://www.nmc-probe.org/wiki/Machines:Kodiak,” Accessed September 5, 2014.

[94] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova, “Scalable in situ scientific data encoding for analytical query processing,” in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2013.

[95] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C. S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, “ISABELA-QA: Query-driven analytics with ISABELA-compressed extreme-scale scientific data,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC’11)*, 2011.

[96] D. Laney, S. Langer, C. Weber, P. Lindstrom, and A. Wegener, “Assessing the effects of data compression in simulations using physically motivated metrics,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[97] J. Lee, M. Winslett, X. Ma, and S. Yu, “Enhancing data migration performance via parallel data compression,” in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, ser. IPDPS ’02, 2002.

[98] R. Li, R. Guo, Z. Xu, and W. Feng, “A prefetching model based on access popularity for geospatial data in a cluster-based caching system,” *Int. J. Geogr. Inf. Sci.*, vol. 26, no. 10, Oct. 2012.

[99] S. Li and H. Huang, “Black-box performance modeling for solid-state drives,” in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, 2010.

[100] T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu, “A dynamically scalable cloud data infrastructure for sensor networks,” in *Proceedings of the 6th Workshop on Scientific Cloud Computing (ScienceCloud)*, 2015.

[101] 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.

[102] T. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, Z. Zhang, and I. Raicu, “A convergence of key-value storage systems from clouds to supercomputer,” *Concurr. Comput. : Pract. Exper.*, 2011 (accepted).

[103] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai, “Efficient batched synchronization in dropbox-like cloud storage services,” in *Proceedings of the 14th International Middleware Conference*, 2013.

[104] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Enabling software management for multicore caches with a lightweight hardware support," in *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009.

[105] N. Liu, C. Carothers, J. Cope, P. Carns, and R. Ross, "Model and simulation of exascale communication networks," *Journal of Simulation*, vol. 6, no. 4, pp. 227–236, 2012.

[106] N. Liu and C. D. Carothers, "Modeling billion-node torus networks using massively parallel discrete-event simulation," in *Proceedings of IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2011.

[107] 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.

[108] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, "Bio-sequence database scanning on a gpu," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, 2006.

[109] 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.

[110] LZO, "http://www.oberhumer.com/opensource/lzo," Accessed September 5, 2014.

[111] K.-L. Ma, "In situ visualization at extreme scale: Challenges and opportunities," *Computer Graphics and Applications, IEEE*, vol. 29, no. 6, 2009.

[112] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-situ processing and visualization for ultrascale simulations," in *Journal of Physics: Conference Series*, vol. 78, no. 1, 2007.

[113] J. P. MacDonald, "File system support for delta compression," University of California, Berkley, Tech. Rep., 2000.

[114] T. Malik, A. Gehani, D. Tariq, and F. Zaffar, "Sketching distributed data provenance," in *Data Provenance and Data Management in eScience*, 2013, pp. 85–107.

[115] A. Manzanares, X. Ruan, S. Yin, J. Xie, Z. Ding, Y. Tian, J. Majors, and X. Qin, "Energy efficient prefetching with buffer disks for cluster file systems," in *Proceedings of the 2010 39th International Conference on Parallel Processing*, 2010.

[116] B. Mao, H. Jiang, D. Feng, S. Wu, J. Chen, L. Zeng, and L. Tian, "HPDA: A hybrid parity-based disk array for enhanced performance and reliability," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010.

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

[118] A. J. McAuley, “Reliable broadband communication using a burst erasure correcting code,” in *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, 1990, pp. 297–306.

[119] D. Meister, A. Brinkmann, and T. Süß, “File recipe compression in data deduplication systems,” in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[120] D. Meister, J. Kaiser, and A. Brinkmann, “Block locality caching for data deduplication,” in *Proceedings of the 6th International Systems and Storage Conference*, 2013.

[121] Message Pack, “http://msgpack.org/,” Accessed December 13, 2014.

[122] Microsoft Azure, “https://azure.microsoft.com,” 2014.

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

[124] G. E. Moore, *Electronics*, vol. 38, no. 8, 1965.

[125] MPEG-1, “http://en.wikipedia.org/wiki/MPEG-1,” Accessed September 5, 2014.

[126] MPICH, “http://www.mpich.org/,” Accessed December 10, 2014.

[127] S. Mu, K. Chen, Y. Wu, and W. Zheng, “When paxos meets erasure code: Reduce network and storage cost in state machine replication,” in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2014.

[128] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns, “Modeling a million-node dragonfly network using massively parallel discrete-event simulation,” in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012.

[129] K.-K. Muniswamy-Reddy, “Foundations for provenance-aware systems,” 2010.

[130] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, “Layering in provenance systems,” in *Proceedings of the 2009 USENIX Annual Technical Conference*, 2009.

[131] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, “Provenance-aware storage systems,” in *Proceedings of the annual conference on USENIX ’06 Annual Technical Conference*, 2006.

[132] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, “Making a cloud provenance-aware,” in *1st Workshop on the Theory and Practice of Provenance*, 2009.

[133] A. Muthitacharoen, B. Chen, and D. Mazières, “A low-bandwidth network file system,” in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[134] 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.

[135] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-aware approach: An incremental checkpoint/restart model in hpc environments," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, 2008.

[136] NetCDF, "http://www.unidata.ucar.edu/software/netcdf," Accessed September 5, 2014.

[137] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[138] 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.

[139] M. Noeth, J. Marathe, F. Mueller, M. Schulz, and B. de Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, 2006.

[140] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley db," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1999.

[141] Open MPI, "http://www.open-mpi.org/," Accessed December 10, 2014.

[142] OpenMP, "http://openmp.org/wp/," Accessed December 9, 2014.

[143] X. Ouyang, S. Marcarelli, and D. K. Panda, "Enhancing Checkpoint Performance with Staging IO and SSD," in *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, 2010.

[144] K. Park, S. Ihm, M. Bowman, and V. S. Pai, "Supporting practical content-addressable caching with czip compression," in *2007 USENIX Annual Technical Conference*, 2007.

[145] A. Parker-Wood, D. D. E. Long, E. L. Miller, M. Seltzer, and D. Tunkelang, "Making sense of file systems through provenance and rich metadata," University of California, Santa Cruz, Tech. Rep. UCSC-SSRC-12-01, Mar. 2012.

[146] S. Patil and G. Gibson, "Scale and concurrency of GIGA+: file system directories with millions of files," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, 2011.

[147] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1988.

[148] J. S. Plank, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications," University of Tennessee, Tech. Rep., 2007.

[149] J. S. Plank, M. Blaum, and J. L. Hafner, "SD Codes: Erasure codes designed for how storage systems really fail," in *Proceedings of the 11th USENIX conference on File and Storage Technologies*, 2013.

[150] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proccedings of the 7th Conference on File and Storage Technologies*, 2009.

[151] S. Podlipnig and L. Böszörmenyi, “A survey of web cache replacement strategies,” *ACM Comput. Surv.*, vol. 35, no. 4, Dec. 2003.

[152] PPL, “http://msdn.microsoft.com/en-us/library/dd492418.aspx,” Accessed December 13, 2014.

[153] Protocol Buffers, “http://code.google.com/p/protobuf/,” Accessed September 5, 2014.

[154] Provenance aware service oriented architecture, “http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome,” Accessed July 6, 2015.

[155] K. Qiao, F. Tao, L. Zhang, and Z. Li, “A ga maintained by binary heap and transitive reduction for addressing psp,” in *Intelligent Computing and Integrated Systems (ICISS), 2010 International Conference on*, Oct 2010.

[156] F. Quaglia and A. Santoro, “Nonblocking checkpointing for optimistic parallel simulation: Description and an implementation,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 6, Jun. 2003.

[157] I. Raicu, I. T. Foster, and P. Beckman, “Making a case for distributed file systems at exascale,” in *Proceedings of the third international workshop on Large-scale system and application performance*, 2011.

[158] 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.

[159] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, “Toward loosely coupled programming on petascale systems,” in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[160] 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.

[161] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay, “Accelerating large-scale data exploration through data diffusion,” in *Proceedings of the 2008 international workshop on Data-aware distributed computing*, 2008.

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

[163] I. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the Society of Industrial and Applied Mathematics*, no. 2, 06/1960.

[164] K. Ren, Q. Zheng, S. Patil, and G. Gibson, “IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion,” in *IEEE/ACM International Conference on Supercomputing*, 2014.

[165] S. Rizvi and T.-S. Chung, “Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems,” in *2nd International Conference on Computer Engineering and Technology (ICCET)*, 2010.

[166] L. Rizzo, “Effective erasure codes for reliable computer communication protocols,” *SIGCOMM Comput. Commun. Rev.*, no. 2, pp. 24–36, Apr.

[167] R. Rodrigues and B. Liskov, "High availability in DHTs: erasure coding vs. replication," in *Proceedings of the 4th international conference on Peer-to-Peer Systems*, 2005.

[168] B. Romoser, Z. Zong, R. Fares, J. Wood, and R. Ge, "Using intelligent prefetching to reduce the energy consumption of a large-scale storage system," in *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*, 2013.

[169] S3FS, "https://code.google.com/p/s3fs/," Accessed March 6, 2015.

[170] N. Santos and A. Schiper, "Achieving high-throughput state machine replication in multi-core systems," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, 2013.

[171] E. R. Schendel, S. V. Pendse, J. Jenkins, D. A. Boyuka, II, Z. Gong, S. Lakshminarasimhan, Q. Liu, H. Kolla, J. Chen, S. Klasky, R. Ross, and N. F. Samatova, "Isobar hybrid compression-i/o interleaving for large-scale parallel i/o optimization," in *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing*, 2012.

[172] 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*, 2002.

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

[174] W. Schwitzer and V. Popa, "Using protocol buffers for resource-constrained distributed embedded systems," Technische Universitaet Muenchen, Tech. Rep., 2011.

[175] SDSS Query, "http://cas.sdss.org/astrodr6/en/help/docs/realquery.asp," Accessed September 5, 2014.

[176] L. Shi, Z. Liu, and L. Xu, "Bwcc: A fs-cache based cooperative caching system for network storage system," in *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, 2012.

[177] 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.

[178] 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.

[179] Snappy, "https://code.google.com/p/snappy/," Accessed September 5, 2014.

[180] Stefan Podlipnig and Laszlo Boszormenyi, "A survey of Web cache replacement strategies," *ACM Computing Surveys (CSUR), Volume 35 Issue 4*, 2003.

[181] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.

[182] SWI-Prolog, "http://www.swi-prolog.org/," Accessed December 7, 2014.

[183] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms.* Prentice Hall; 2nd edition, 2006, pp. 531–532.

[184] 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.

[185] F. Tao, K. Qiao, L. Zhang, Z. Li, and A. Nee, "GA-BHTR: an improved genetic algorithm for partner selection in virtual manufacturing," *International Journal of Production Research*, vol. 50, no. 8, 2012.

[186] A. Tikotekar, G. Vallee, T. Naughton, S. L. Scott, and C. Leangsuksun, "Evaluation of fault-tolerant policies using simulation," in *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, 2007.

[187] Titan, "https://www.olcf.ornl.gov/titan/," Accessed December 10, 2014.

[188] Top500, "http://www.top500.org/list/2014/06/," Published June 2014; Accessed September 5, 2014.

[189] TPCH Benchmark, "http://www.tpc.org/tpch," 2014.

[190] M. Voss and R. Eigenmann, "Reducing parallel overheads through dynamic serialization," in *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, ser. IPPS '99/SPDP '99, 1999.

[191] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, "Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.

[192] G. Wang, X. Liu, A. Li, and F. Zhang, "In-memory checkpointing for mpi programs by xor-based double-erasure codes," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2009.

[193] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proceedings of IEEE International Conference on Big Data (BigData Conference)*, 2014.

[194] 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.

[195] D. Warneke and O. Kao, "Nephele: Efficient parallel data processing in the cloud," in *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, ser. MTAGS '09, 2009.

[196] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002, pp. 328–338.

[197] 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.

[198] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

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

[200] C. Wu, X. He, Q. Cao, C. Xie, and S. Wan, "Hint-k: An efficient multi-level cache using k-step hints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, 2013.

[201] X. Wu and A. L. N. Reddy, "SCMFS: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[202] P. Xia, D. Feng, H. Jiang, L. Tian, and F. Wang, "Farmer: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance," in *Proceedings of the 17th international symposium on High performance distributed computing*, 2008.

[203] G. Xu, S. Lin, G. Wang, X. Liu, K. Shi, and H. Zhang, "Hero: Heterogeneity-aware erasure coded redundancy optimal allocation for reliable storage in distributed networks," in *Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International*, 2012.

[204] G. Xu, S. Lin, H. Zhang, X. Guo, and K. Shi, "Expander code: A scalable erasure-resilient code to keep up with data growth in distributed storage," in *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*, 2013.

[205] Y. Xu, C. Xing, and L. Zhou, "A cache replacement algorithm in hierarchical storage of continuous media object," in *Advances in Web-Age Information Management: 5th International Conference*, 2004.

[206] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of SSD and HDD," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[207] S. Yin, Y. Tian, J. Xie, X. Qin, X. Ruan, M. Alghamdi, and M. Qiu, "Reliability analysis of an energy-aware raid system," in *Proceedings of the 30th IEEE International Performance Computing and Communications Conference*, 2011.

[208] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008.

[209] J. Yue, Y. Zhu, Z. Cai, and L. Lin, "Energy and thermal aware buffer cache replacement algorithm," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[210] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[211] I. Zecena, M. Burtscher, T. Jin, and Z. Zong, "Evaluating the performance and energy efficiency of n-body codes on multi-core cpus and gpus," in *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*, 2013.

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

[213] D. Zhan, H. Jiang, and S. C. Seth, "Locality & utility co-optimization for practical capacity management of shared last level caches," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012.

[214] X. Zhang, K. Davis, and S. Jiang, "iTransformer: Using SSD to improve disk scheduling for high-performance I/O," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.

[215] X. Zhang, L. Ke, K. Davis, and S. Jiang, "iBridge: Improving unaligned parallel file access with solid-state drives," in *Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium*, 2013.

[216] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, and M. Wilde, "Design and evaluation of a collective i/o model for loosely- coupled petascale programming," in *IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, ser. MTAGS '08, 2008.

[217] 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.

[218] Z. Zhang and S. Fu, "Macropower: A coarse-grain power profiling framework for energy-efficient cloud computing," in *IEEE International Performance Computing and Communications Conference*, 2011.

[219] Z. Zhang, Q. Guan, and S. Fu, "An adaptive power management framework for autonomic resource configuration in cloud computing infrastructures," in *Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International*, 2012.

[220] 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.

[221] D. Zhao, N. Liu, D. Kimpe, R. Ross, X.-H. Sun, and I. Raicu, "Towards exploring data-intensive scientific applications at extreme scales through systems and simulations," *Parallel and Distributed Systems, IEEE Transactions on*, 2015 (accepted).

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

[223] ——, "Towards cost-effective and high-performance caching middleware for distributed systems," *International Journal of Big Data Intelligence*, 2015.

[224] D. Zhao, K. Qiao, J. Yin, and I. Raicu, "Dynamic virtual chunks: On supporting efficient accesses to compressed scientific data," *Services Computing, IEEE Transactions on*, 2015 (minor revision).

[225] 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.

[226] ——, “HyCache: A user-level caching middleware for distributed file systems,” in *Proceedings of IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013.

[227] 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.

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

[229] ——, “Incremental isometric embedding of high-dimensional data using connected neighborhood graphs,” *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, vol. 31, no. 1, Jan. 2009.

[230] D. Zhao, X. Yang, I. Sadooghi, G. Garzoglio, S. Timm, and I. Raicu, “High-performance storage support for scientific applications on the cloud,” in *Proceedings of the 6th Workshop on Scientific Cloud Computing (ScienceCloud)*, 2015.

[231] 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.

[232] 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.

[233] 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.

[234] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, “FusionFS: Toward supporting data-intensive scientific applications on extreme-scale distributed systems,” in *Proceedings of IEEE International Conference on Big Data*, 2014.

[235] Y. Zhao, X. Fei, I. Raicu, and S. Lu, “Opportunities and challenges in running scientific workflows on the cloud,” in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, 2011.

[236] 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.

[237] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao, “Efficient querying and maintenance of network provenance at internet-scale,” in *Proceedings of the 2010 international conference on Management of data*, 2010, pp. 615–626.

[238] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/o-aware batch scheduling for petascale computing systems," in *Cluster Computing, IEEE International Conference on*, 2015.

[239] Z. Zhu and X. Zhang, "Access-mode predictions for low-power cache design," *IEEE Micro*, vol. 22, no. 2, Mar. 2002.