1-15-2016

Optimizations for Energy-Aware, High-Performance and Reliable Distributed Storage Systems

Cengiz Karakoyunlu
University of Connecticut, cengiz.k@uconn.edu

Optimizations for Energy-Aware, High-Performance and Reliable Distributed Storage Systems

Cengiz Karakoyunlu, Ph.D.

University of Connecticut, 2016

ABSTRACT

With the decreasing cost and wide-spread use of commodity hard drives, it has become possible to create very large-scale storage systems with less expense. However, as we approach exabyte-scale storage systems, maintaining important features such as energy-efficiency, performance, reliability and usability became increasingly difficult. Despite the decreasing cost of storage systems, the *energy consumption* of these systems still needs to be addressed in order to retain cost-effectiveness. Any improvements in a storage system can be outweighed by high energy costs. On the other hand, large-scale storage systems can benefit more from the object storage features for improved *performance* and *usability*. One area of concern is metadata performance bottleneck of applications reading large directories or creating a large number of files. Similarly, computation on big data where data needs to be transferred between compute and storage clusters adversely affects I/O performance. As the storage systems become more complex and larger, transferring data between remote compute and storage tiers becomes impractical. Furthermore, storage systems implement *reliability* typically at the file system or client level. This approach might

not always be practical in terms of performance. Lastly, object storage features are usually tailored to specific use cases that makes it harder to use them in various contexts.

In this thesis, we are presenting several approaches to enhance energy-efficiency, performance, reliability and usability of large-scale storage systems. To begin with, we improve the *energy-efficiency* of storage systems by moving I/O load to a subset of the storage nodes with energy-aware node allocation methods and turn off the unused nodes, while preserving load balance on demand. To address the metadata *performance* issue associated with large creates and directory reads, we represent directories with object storage collections and implement *lazy creation* of objects. Similarly, in-situ computation on large-scale data is enabled by using object storage features to integrate a computational framework with the existing object storage layer to eliminate the need to transfer data between compute and storage silos for better *performance*. We then present parity-based redundancy using object storage features to achieve *reliability* with less performance impact. Finally, *unified storage* brings together the object storage features to meet the needs of distinct use cases; such as cloud storage, big data or high-performance computing to alleviate the unnecessary fragmentation of storage resources. We evaluate each proposed approach thoroughly and validate their effectiveness in terms of improving energy-efficiency, performance, reliability and usability of a large-scale storage system.

Optimizations for Energy-Aware, High-Performance and Reliable Distributed Storage Systems

Cengiz Karakoyunlu

B.S., Worcester Polytechnic Institute, 2010

M.S., University of Connecticut, 2013

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2016

APPROVAL PAGE

Doctor of Philosophy Dissertation

Optimizations for Energy-Aware, High-Performance and Reliable Distributed Storage Systems

Presented by

Cengiz Karakoyunlu, B.S., M.S.

Major Advisor _____
Dr. John A. Chandy

Associate Advisor _____
Dr. Bing Wang

Associate Advisor _____
Dr. Mohammad Khan

University of Connecticut

2016

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help of so many people. First and foremost, I would like to show my sincere gratitude to my advisor, Prof. John A. Chandy, who has supported and guided me throughout my PhD studies. Moreover, I would like to thank Prof. Bing Wang, Prof. Mohammad Khan and Prof. Mark M. Tehranipoor for advising and assisting me during my PhD.

I must add my deepest appreciation to my wife, Bilge, for her unlimited support since the beginning of my PhD studies. Completing my PhD would have been much harder without her help. I also would like to thank my brother, Deniz, and my parents, Semiha and Ahmet, who supported me all my life.

Last but not least, I would like to thank all my past and present group members; Orko Momin, Paul Wortman, Rohit Mehta and Michael Runde for their help. I am also thankful to Dries Kimpe and Phil Carns for their contributions during my internship at Argonne National Laboratory and Alma Riska for her assistance during my internship at EMC.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The cost of disk arrays has decreased in the last decades and many applications today use large-scale storage systems that consist of vast numbers of cheap disk arrays. The primary purpose of these systems is to achieve high performance with reasonable cost under concurrent accesses from various users and workloads. These systems typically manage data by using clustered file systems [45, 131, 40, 115] that achieve high I/O performance by striping data across nodes and accessing data in parallel. However, as the size of data managed by the storage systems and the number of concurrent users increases, these systems become inefficient due to commonly observed problems with regards to energy efficiency, performance, reliability and usability.

One of the most critical problems observed in storage systems is the high energy consumption. Energy efficiency has attracted considerable interest from storage system developers in recent years, since the increasing energy consumption and

maintenance costs of storage systems have started to limit any possible performance gain with increasing scale. As an example, according to [77], the estimated energy consumption of U.S. data centers was more than 100 billion kilowatt-hours in 2011. Between 2005 and 2010, data centers consumed between 1.7% and 2.2% of all electricity in the U.S. [80], matching the energy consumption of the aviation industry. Increasing energy consumption also means higher cooling costs and additionally, a storage system with an underperforming cooling mechanism may have further reliability issues. It is, therefore, critical to have an energy-efficient storage system not only for reduced energy costs; but, also to meet reliability and performance goals.

In addition to that, although storage systems distribute I/O load equally across available nodes, they suffer from switch limitations and incast [103, 94]. As a result of the incast behavior, there is a limit to the number of storage nodes (e.g. 4 servers in a cluster-based storage network as in [81]) across which data can be striped for parallel access. Beyond this limit, the I/O performance no longer scales and in fact deteriorates. Therefore, it is critical to develop energy-efficient node allocation techniques in large-scale storage systems. We exploit the heterogeneity in the user metadata for energy-aware storage node allocation. As an example, if a storage system consists of N storage nodes and M of them max out the incast bandwidth ($N > M$), then each user can be assigned a separate subset of M storage nodes based on certain metadata (i.e. user id, usage pattern) and any storage node that is not allocated for any user can be switched into low power modes. In this thesis, we propose several methods to map subsets of storage nodes to different users while achieving uniform system utilization on demand and evaluate them with simulations using real workloads. We specifically show how these methods can be implemented in a cloud storage system.

On the other hand, there also have been some challenges in large-scale storage sys-

tems in terms of performance; with respect to both metadata and data operations. These systems typically employ a single metadata server that will be a performance bottleneck when the system scales. A cluster consisting of 25000 nodes can simultaneously create more than half a million files to store checkpointing data [100]. There are existing approaches to distribute metadata load more evenly across a storage system [129, 55]; however, there is a need to implement even better approaches as the systems grow in size. We show how object storage features can be used to improve metadata performance with respect to two common metadata-intensive operations; reading multiple entries in a large directory and creating large number of files in a single directory. Our implementation is based on a parallel file system integrated with object-based storage, Ohio Supercomputing Center's PVFS-OSD implementation [29].

Similarly, data performance is also of critical importance in large-scale storage systems as high-performance computing on big data has become a common use case [12, 4, 19]. Clusters and cloud storage applications that perform computation on big data typically employ separate compute and storage clusters, as the requirements of the compute and storage tiers are different from each other. A serious drawback of this architecture is the need to move large amounts of data from the storage nodes to the compute nodes in order to perform computation and then move the results back to the storage cluster. Considering the fact that storage systems are projected to reach exabyte scale in the near future [63]; moving large chunks of data between storage and compute nodes is highly inefficient. As part of this thesis, we show how a computation framework can benefit from the features of the underlying object storage to enable in-situ data analytics capabilities. We propose an example of this approach by implementing a MapReduce framework (Hadoop [5]) on top of an object storage

system (Ceph [129]) and evaluate it with various benchmarks.

Reliability is another important aspect of large-scale storage systems. In situations where cost or performance is of primary concern, reliability might be overlooked; but, there might be use cases involving critical or time-sensitive data where data reliability is a must. Replication is commonly used to achieve high reliability; however, storage space overhead can make replication an inefficient way of protecting data. On the other hand, parity-based techniques utilize much less storage space compared to replication. Previous studies implemented parity-based redundancy techniques at the file system or client level [71, 95]. These approaches do not take advantage of the existing intelligence in object storage and additionally clients might not always be available to implement redundancy techniques. We propose a parity-based redundancy scheme on object storage, using OSC-OSD object storage emulation [54] and evaluate it under various use cases using PVFS clients [45].

Finally, we consider the problem of object storage implementations [129, 36, 40, 131] being tailored to specific use cases or data models that makes it difficult to reuse them in various tasks (i.e. big data, cloud storage or high-performance computing (HPC) storage tasks). As a result, management overhead and storage provisioning for tasks with diverse storage needs increases. In this thesis, we first identify some of the most popular large-scale data models in use today and identify core requirements for each and then propose a new object storage API, Advanced Storage Group (ASG) interface, that seeks to unify features necessary to support these data models without compromising usability or flexibility. A number of case studies are presented that evaluate how the proposed API would be used as a foundation for a diverse set of storage architectures.

1.2 Thesis Outline

The rest of this thesis is structured as follows; in Chapter 2, we first introduce object storage followed by an overview of clustered file systems. Chapter 3 proposes techniques to have an energy-aware cloud storage system while at the same time trying to achieve uniform system utilization on demand. Chapter 4 describes methods to tackle the metadata performance problem in large-scale storage systems and Chapter 5 presents a data analytics framework on object storage to perform in-situ MapReduce computation on existing data. In Chapter 6, an approach to implement parity-based redundancy using object storage framework is presented. Chapter 7 introduces a unified object storage system. In Chapter 8, we conclude the thesis and discuss future directions of our research.

Chapter 2

Background

This chapter first presents an introduction to object storage followed by a brief overview of clustered file systems.

2.1 Object-based Storage

Object-based storage forms the basis of all studies included in this thesis; except for the energy-aware storage utilization in Chapter 3. While we implemented the energy-aware storage utilization methods in Chapter 3 for a cloud storage environment, these methods are also applicable to object-based storage systems. Before we delve into the details of the studies included in this thesis, it is necessary to provide a background on object-based storage.

Object-based storage [67, 89] came out as an alternative storage model to the traditional block-based model, which does not fully exploit the intelligence available at the storage units. It rapidly became a commonly used architecture for referencing

and accessing data distributed over large numbers of storage devices in these systems. Network-Attached Secure Disk (NASD) [67] is the primary work on object-based storage. NASD introduces variable-length objects with attributes, rather than fixed-length traditional blocks, to enable self-management and to obviate the need to know about the host operating system. Moving data management to the storage disks increases the networking, security and space management capabilities.

Object-based storage stores and accesses data as ordered logical collection of bytes in flexible-sized discrete containers, called *objects*, instead of using the traditional fixed-sized, block-based containers. Objects can have variable sizes and each object has a unique numerical identifier.

Object-based storage has been used by several high-performance distributed file systems [40, 94, 115, 129, 45] where data was striped and stored in object-based storage servers and metadata was stored in metadata servers. Devulapalli et al. has also studied integrating a standards-based object-storage with PVFS and showed that object attributes can be used to improve the performance of metadata operations. [54, 56, 28].

The Ohio Supercomputing Center looked at mapping Parallel Virtual File System [45] on top of an existing object-based storage emulation [54, 56]. This mapping moved the functionality of the common components of a traditional storage system, such as I/O, directory, or metadata servers, to OSDs and improved the performance of the overall system thanks to the capabilities of the object-based storage devices [28, 29, 53].

Several cloud storage systems have been implemented on object-based storage architectures [2, 13, 19]. Some companies implemented their own object-based storage systems for specific use cases [36].

Developing interest in the object-based storage has led to the creation of the object-based storage device (OSD) standard [124] and object-based storage has been used as a foundation of the numerous types of systems. Although the object-based storage model was originally designed as a device-level interface, today's large-scale storage systems more commonly repurpose the object model as a software interface atop a variety of storage substrates [54, 56, 18, 130, 40, 84]. Seagate has recently introduced an object-based storage platform called Kinetic [116] and more general enterprise-level object storage systems have also been developed [8, 9, 20].

According to the T10 object-based storage standard [124], each object stores data and data attributes that can be controlled by the user. Data attributes can be used to store metadata describing the data (i.e. size, name, replica locations, QoS attributes and device-managed metadata such as security information etc.) and metadata management operations to query these attributes can be offloaded from dedicated servers to object storage for improved performance [29, 89, 131].

As a result, object-based storage enables offloading data management operations from the client to the storage system and it increases the interaction between the storage system and the end-user. It simplifies the data management of a storage system and improves its performance. Object-based storage also allows considerable flexibility for a variety of higher-level data models to be built on top of it. And it closes the gap between computation and storage units.

2.2 Clustered File Systems

File systems are software abstractions built on top of storage devices acting as intermediate layers for applications and clients accessing these devices. While some file systems can be local to a physical node (EXT4, FAT32 etc.), clustered file systems are mounted on multiple physical nodes simultaneously. A clustered file system node can operate on data individually in addition to sharing and transferring it to other nodes in the system through a network architecture. Each clustered file system node can have unique properties that sets it apart from other nodes in the system, but the clustered file system itself acts as a whole and maintains the storage nodes. As a result, clustered file systems increases the scalability of storage systems and offloads the management tasks from individual nodes.

Parallel file systems are the most commonly used examples of clustered file systems, storing and accessing data in a network of storage nodes, as shown in Figure 2.2.1. The most important feature of the parallel file systems is striping of data across the storage nodes for parallel and concurrent access. Data striping also enables parity-based redundancy schemes and improves the reliability of the system. Another important feature of the parallel file systems is the separation of data operations from metadata operations. Clients contact metadata servers to fetch information describing data they want to access, i.e. size, permissions, stripe width etc. Clients can then directly access the storage nodes for I/O operations with the metadata information fetched from the metadata servers. This architecture improves the scalability and performance of the system and enables parallel, direct and concurrent access to data from multiple clients.

Commonly used implementations of distributed parallel file systems include Par-

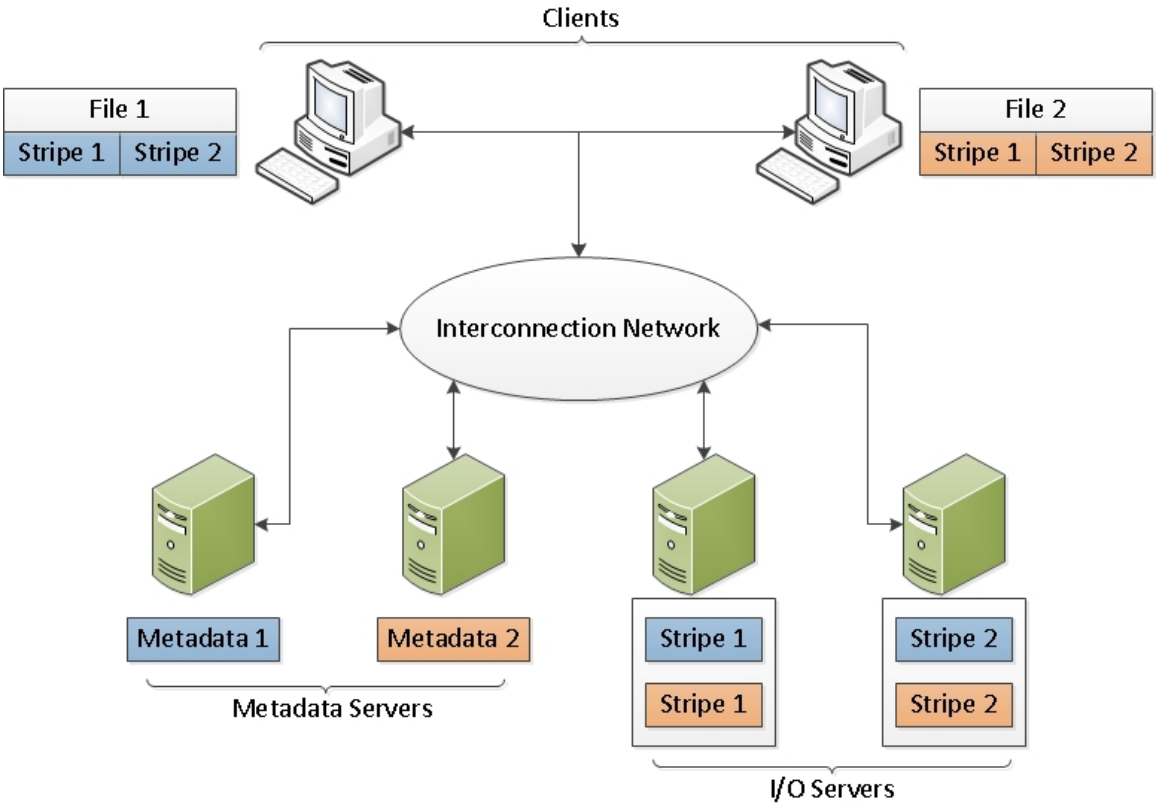


FIGURE 2.2.1: Parallel storage architecture

allel Virtual File System (PVFS) [45], Ceph [129], IBM's General Parallel File System (GPFS) [115], Google File System (GFS) [66], Hadoop Distributed File System (HDFS) [120], Lustre [40], Isilon OneFS [10] and Panasas ActiveStor File System (PanFS) [131]. Most of these systems are based on the object-based storage architecture that was introduced in Section 2.1. As discussed previously, objects consist of data and attributes that store metadata and that can be accessed individually. Separation of data and metadata at the object level fits the architecture of parallel file systems and improves the scalability and reliability of storage systems.

Chapter 3

Energy Aware Cloud Storage Utilization

Cloud systems have gained popularity over the last decade by enabling users to store data, host applications and perform computations over the network. Cloud systems significantly decrease the cost on the user end as management, maintenance and administration tasks are typically handled by the cloud providers. Cloud providers also benefit from this scheme as they can utilize system resources more efficiently through techniques, such as virtualization, enabling them to achieve better performance and energy efficiency. There are numerous cloud providers offering a broad range of services [12, 4, 19].

Even though cloud systems tend to have lower energy costs compared to traditional HPC clusters due to better utilization techniques, the increasing energy consumption of cloud systems still needs to be addressed as the amount of data stored and the number of computations and applications in cloud increase steadily. As mentioned in Chapter 1.1, energy consumption of data centers increase steadily and it is critical to

implement energy-efficient data management mechanisms in these systems.

A typical cloud system is shown in Figure 3.0.1. In this chapter, we refer to any application using the back-end storage of a cloud system as the *user* of that storage system. There are several components in a cloud system contributing to the overall energy consumption - namely processing units, network components and storage systems. In this thesis, we specifically target the energy consumption of the cloud storage infrastructure forming the back-end of the cloud computing units, since the storage system costs constitute an important fraction (between 25-35%) of overall cloud system costs [70, 78, 61]. Storage systems also have more idle periods since data stored is usually redundant and archival, written once and not touched again [91]. There have been many studies to reduce the energy consumption of other components of cloud systems. However, since idleness is not usually available in the network and processing units of clouds, these studies have been mostly on virtual machine consolidation, workload characterization, data migration or scheduling. In this chapter of the thesis, we propose methods to have an energy-aware cloud storage system while at the same time trying to achieve uniform system utilization on demand. In particular, we take advantage of idleness existing in cloud storage systems and try to switch inactive nodes into low power modes. Our approach is driven by two key assumptions: first, the cloud storage system suffers from incast, and second, most of the data stored in the cloud (as much as 75%) is not heavily accessed [61], creating idle periods. It is important to note that our methods can also be implemented in the storage systems that form the back-end of computational platforms (i.e. Hadoop clusters), where there might be idle periods [76, 96].

Incast is a condition that occurs because of queue limitations in most network switches [94]. As mentioned previously, incast limits the number of nodes across

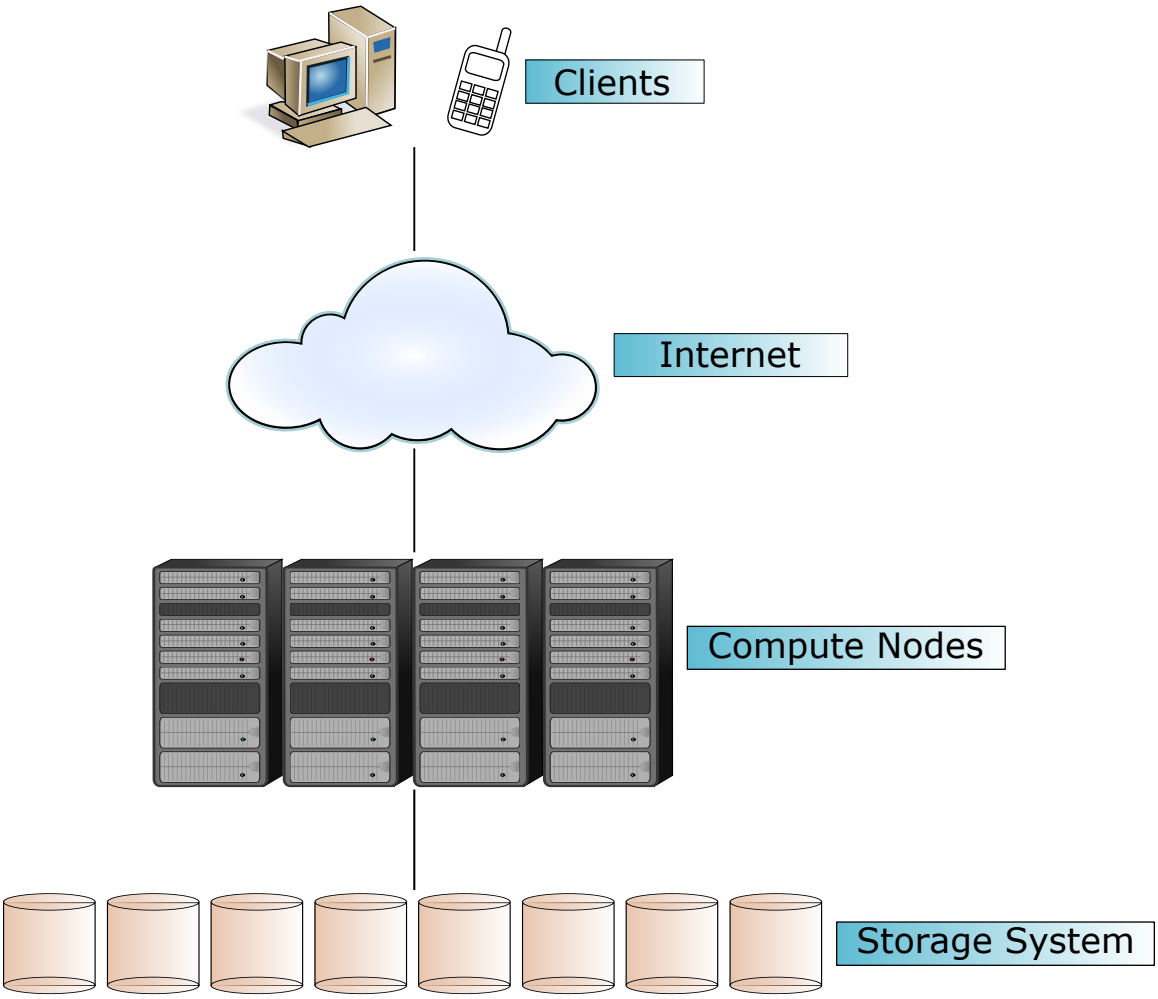


FIGURE 3.0.1: Typical cloud system architecture

which data can be striped and performance loss occurs if data is striped on a number of nodes that is greater than this limit. If M represents the number of nodes at which the performance maxes out, from a performance point of view, there is no point in using more than M nodes to increase parallelism. While for performance scaling, we would only need M storage nodes in the system, however, for storage capacity reasons, we may need more than M nodes. In such systems, we would need to keep these extra nodes active and thus waste energy since the network resources are maxed-out by a *subset* (of size M) of the storage nodes.

To save energy, one could turn off these extra storage nodes and activate them when necessary. The difficulty comes while trying to identify which storage nodes should be turned on or off. Our approach is to distribute cloud users across the storage nodes, such that each user is allocated only M nodes - i.e. the limit at which performance is maxed out. Grouping data on a subset of the storage nodes and putting the remaining nodes into low power modes has been studied in many related studies [76, 77, 83]. However, the majority of these studies used data classifications, redundant data or a hot-cold zone approach to group data on a subset of the storage nodes. Our approach is different from existing studies in that we try to group *cloud users* on a subset of the storage nodes without any data classification. Not classifying data enables our methods to be implemented in cloud storage systems with any kind of redundancy scheme. It is important to also note that, because of the job processing structure (batch mode or intelligent schedulers) in clouds, we can effectively turn on and off storage nodes based on the user job submissions since we know a priori which M nodes are going to be used at any time. Therefore, any latency due to transitioning a storage node from inactive to active mode can be hidden.

The challenge becomes now how to allocate a subset of cloud storage nodes for

each user in order to reduce the energy footprint, while at the same time trying to preserve uniform distribution of the system resources if demanded. We take advantage of the heterogeneity in the user metadata to perform storage node allocation for each user: M storage nodes out of N total nodes are allocated for each user based on a certain metadata (user id, usage pattern etc.) and any unallocated or inactive storage node is switched to low power modes. This chapter presents and evaluates numerous approaches to allocate subsets of storage nodes for cloud users. These approaches build extensively on preliminary methods we presented in our earlier work [73].

The rest of this chapter is organized as follows: Section 3.1 introduces related studies in the research area. Section 3.2 describes the implementation of the node allocation methods we are proposing. Section 3.3 presents a mathematical model to estimate the outcomes of the proposed methods. Section 3.4 evaluates the proposed methods theoretically to show their approximation factors to the optimal solutions. Section 3.5 gives the experimental evaluation results of the proposed node allocation methods and validates the mathematical model proposed in Section 3.3. Finally, we present conclusions and possible future work in Section 3.6.

3.1 Related Work

There have been a large number of studies aiming at reducing the energy consumption of distributed systems. The energy consumption of cloud systems has been studied extensively by the research community. Srikantaiah et al. reduce the energy consumption of cloud systems with workload consolidation while trying to find optimal performance energy trade-off points [122]. In [79], authors present energy aware pro-

visioning of virtual machines in a cloud system. Harnik et al. investigate how cloud storage systems can operate at low-power modes while maximizing data availability and the number of nodes to power down [70]. Duy et al. propose a green scheduling algorithm to predict the future load in a cloud system and to turn off unused nodes [58]. CloudScale reserves resources based on usage in a multi-tenant cloud to reduce energy consumption [119]. Rabbit is a distributed file system trying to save energy by turning off nodes while making sure that at least primary data replicas are available [30]. Beloglazov et al. present a model to detect an overloaded host and dynamically reallocate the virtual machines on that host for improved energy efficiency and performance [37]

There also have been numerous studies to reduce the energy consumption of storage and file systems in general. Most of the existing energy saving techniques for these systems attempt to move less frequently used data to a subset of the nodes. Massive Array of Idle Disks [48] (MAID) forms two groups of storage nodes in the system - *active* and *passive*. New requests are typically handled by the active nodes, and if not, they are forwarded to the passive nodes. The performance of MAID, however, is dependent on the workload and cache characteristics. Popular Data Concentration [104] (PDC) is another similar technique where frequently accessed data is migrated to a group of storage nodes, called *active* nodes. Before migrating data, PDC needs to predict the future load for each storage node. Although performing better than MAID for small workloads, PDC suffers from the overhead of data migration and load prediction. Wildani et al. present a technique that identifies and brings together data blocks in a workload for better energy management, based on the likelihood of related access [133]. GreenHDFS uses a hot&cold zone approach, where frequently accessed data is located on the storage nodes in the hot zone and

unpopular data is located on the storage nodes in the cold zone [76]. Lightning is an energy-aware cloud storage system that divides the storage nodes into hot&cold zones with data-classification driven data placement [77]. The purpose of dividing the storage nodes into logical hot&cold zones is to increase the idleness in the storage system. There have been other relevant studies that aimed directly at making better use of idle periods in a storage system. Mountroidou et al. presents a framework that identifies when and for how long to activate a power-saving mode to meet given performance&power constraints [92]. They also propose adaptive workload shaping to make use of the idle periods in a workload better [93]. Write-offloading technique shows that enterprise workloads have idle periods as well and these periods can be increased further by offloading writes on spun-down disks to persistent storage [96]. SRCMap is another technique where the workload is selectively consolidated on a subset of storage nodes, proportional to the I/O workload [127]. These data-classification driven placement techniques work well only if one is able to predict data usage and idle period with reasonable accuracy.

Hardware based techniques can also help with energy utilization but is not broadly applicable. Barroso et al. proposed that server components, particularly memory and disk subsystems, need improvements to consume power proportional to their utilization levels [35]. Hibernator uses disks that can operate at different speeds to reduce energy consumption while trying to meet performance goals [139]. Sheikh et al. present a power-aware server selection method for nano data centers [118]; however, their solution is request oriented and they assume all files are available on all data servers.

Architectural or file system optimizations present another opportunity to save energy. Ganesh et al. [62] has shown that the Log Structured File System (LFS) can be

used to reduce energy consumption, since the write requests are recorded in a log file making it possible to know on the client side which storage node will handle the write request. This approach suffers from the overhead of cleaning the log file. Leverich and Kozyrakis present a technique to reduce the energy consumption of Hadoop clusters using covering subsets to ensure data availability [83]. They find a trade-off between energy savings and overall performance of the system. Pergamum is a distributed archival storage system that saves energy by avoiding centralized controllers [123]. Zhu et al. proposes power-aware storage cache management algorithms to keep the disks in low-power modes for longer [140]. Diverted Access is another technique that exploits the redundancy in the storage systems to reduce energy consumption [105].

In more recent related studies, Chen et al. present the *k-out-of-n computing* framework [46] with the goal of increasing fault-tolerance and energy-efficiency during storage system access and data processing. Even though, the random and greedy approaches used during the evaluations is similar to the methods we propose, this work is tailored for mobile devices in a dynamic network and unlike our study, it is not concerned with load balancing. In [49], the authors propose a fuzzy logic approach that tries to improve the energy-efficiency of Bluetooth Low Energy (BLE) network used in many Internet-of-Things environments, by predicting sleeping periods of devices in BLE network using their battery levels and throughput-to-workload ratios. Even though, the scope and parameters of this work is completely different than our approach, the authors show a method to benefit from idleness in a system using data from system components. Finally, Sallam et al. present a proactive workload manager that tries to avoid bursty loads and underutilization of resources that might be caused by a reactive workload manager in a virtual environment [114]. They proactively predict the future state of virtual machines by analyzing the recently observed

patterns. This approach is similar to the future prediction method we propose in Dynamic Greedy and Correlation-Based schemes; although, it is tailored for virtual environments.

To the best of our knowledge, there has not been any related study trying to reduce energy consumption in a cloud storage system by using user metadata. The most relevant to our approach is by Wildani et al. to group semantically-related data across the same set of devices to reduce the number of disk accesses resulting in disk spin-ups. However, they group related data, while we group related users together [132].

3.2 Proposed Techniques

In order to tackle the energy efficiency problem of cloud storage systems, we propose allocating nodes for each user based on the metadata information of that user and switch the inactive nodes to low energy modes. As described earlier, we assume a use scenario where a subset of the storage nodes max out the available bandwidth of the system due to incast. Therefore, it is not feasible to allocate more than M nodes to each user both for performance and energy efficiency reasons. Here M represents the number of storage nodes in a subset allocated for a user. Therefore, the problem we are trying to solve is how to map subsets of the storage nodes to the users. Since user metadata heterogeneity is well-defined in large-scale cloud systems, we retrieve user metadata information (i.e. user id, usage pattern) and allocate subsets of storage nodes to the users using this information.

In this chapter, we propose three different methods to map cloud storage nodes

to the cloud users, summarized as follows:

- *Fixed Scheme*: There are four node allocation techniques in this method: *balancing*, *sequential*, *random* and *grouping* and once one of these techniques is chosen for a cloud storage system, it remains in effect unless manually changed.

- *Dynamic Greedy Scheme*: This method extends the *Fixed Scheme* method by periodically doing dynamic reallocation among one of the four allocation techniques (*balancing, sequential, random, grouping*) depending on their costs. The cost of each technique is calculated based on how important it is for the cloud storage system to save energy or to balance load.

- *Correlation-based Scheme*: This method monitors user activities and tries to allocate the same subset of storage nodes to the users who tend to use the cloud storage system concurrently.

The energy-aware node allocation methods proposed here are designed to work for a traditional distributed storage system architecture, but they could also work in a disk array. The proposed methods not only reduce energy consumption, but also balance load on storage nodes depending on metrics selected by the cloud administrators.

Before describing each node allocation method in more detail, we first list some of the usage assumptions in our system and then explain two common features of the node allocation methods: *inactivity threshold* and *job overlapping*.

We assume that;

- All storage nodes are initially off and a storage node is started up as soon as a job arrives.

- A user uses the same amount of storage space on each of its allocated nodes.
- Any job run by a user is divided into equal tasks on the nodes allocated for that user and these tasks are executed concurrently, meaning that they start and complete simultaneously.
- If a user is transferred from a subset of the storage nodes to another subset of the storage nodes, this includes transferring only data. Any jobs that were executed in the old subset of the storage nodes still belong to those nodes.

Inactivity Threshold The node allocation methods we are proposing try to save as much energy as possible and also balance the load on storage nodes depending on metrics selected by the cloud administrators. If a storage node is not allocated for any user at all, then it will stay in a low-energy mode. On the other hand, if a storage node is allocated for one or more users, that node will only switch to a low-energy mode after all jobs using that node are completed. In an HPC system, the completion time of jobs can be predicted because of job scheduling systems. Thus, we can decide when to switch a node to a low-energy mode. However, in a cloud storage system, this predictability is not necessarily always possible and we need to have a condition to decide when to switch a storage node to a low-energy mode. We call this metric the *inactivity threshold*, which can be defined as *the period of time a node continues to operate at full capacity after the completion of the most recent storage system job.* This means that, once a node stays inactive for longer than the inactivity threshold, it can be switched to a low-energy mode. The *inactivity threshold*, in this sense, is similar to *break-even time* in previous studies, which ensures that the energy saved by turning off a node is greater than the energy consumed while switching that node

from active to low-power modes.

In our approach, we define the low-energy mode as the state where a node is completely turned off. However, some modern hard drives have the ability to operate at various speeds [69] thus providing different levels of energy utilization. Therefore, in order to conserve energy it is not mandatory to completely turn off a node. Depending on how much energy saving is demanded by the user, the node allocation techniques can switch nodes into low-speed operating modes without turning them off. In this chapter, in order to show the effect of the node allocation methods better, we assume the worst case and turn off a storage node in low-energy mode.

If a user is allocated to a node that has been turned off, then that node needs to start operating at full capacity again. In this case, the user can not immediately access that storage node, since it will take some time for that node to run at full capacity again. We define the time it takes to start up a completely turned-off node as the *startup time*. If a storage node has been inactive between two jobs for longer than the *inactivity threshold*, then the next job on that node will have to wait for the node to start up.

The *inactivity threshold* and the *startup time* are important parameters in determining how a real cloud storage system is going to be affected by the node assignment methods we are proposing.

Job Overlapping In a large-scale cloud storage system, each storage node may be used by multiple concurrent users. Our node allocation methods might allocate a node to multiple users. Therefore, concurrent users can simultaneously run jobs that use a particular storage node. When two or more jobs overlap on a node, we

assume that node still consumes the same energy per unit time [126]. In other words, we assume the increased activity due to the multiple jobs will not increase energy consumption significantly. This is, for the most part, true because most of the energy is consumed by spinning the drives not by moving actuators on the drive.

3.2.1 Fixed Scheme

In this first scheme, one of the four node allocation techniques described below (balancing, sequential, random or grouping) is chosen by the cloud storage system administrator and is kept fixed unless manually changed. All techniques have one goal in common - exploiting user metadata to allocate storage nodes for users. Individually, each technique performs differently in terms of uniformly allocating storage nodes. These techniques are each described numerically in Figure 3.2.1 with examples.

Balancing Technique

The primary goal of this technique is to balance the load across the cloud storage nodes. We define the *load* here in two different ways:

- The amount of data stored on each node.
- The total time each node stays on serving user requests.

We call balancing the amount of data stored on each node *storage space balancing*. Similarly, balancing the total time each node stays on is called *on-time balancing*. These two balancing techniques ensure that all the nodes in the system are used equally. Otherwise, the simplest energy saving technique is to simply put all users and data on the same subset of nodes and permanently turn off all other nodes. This,

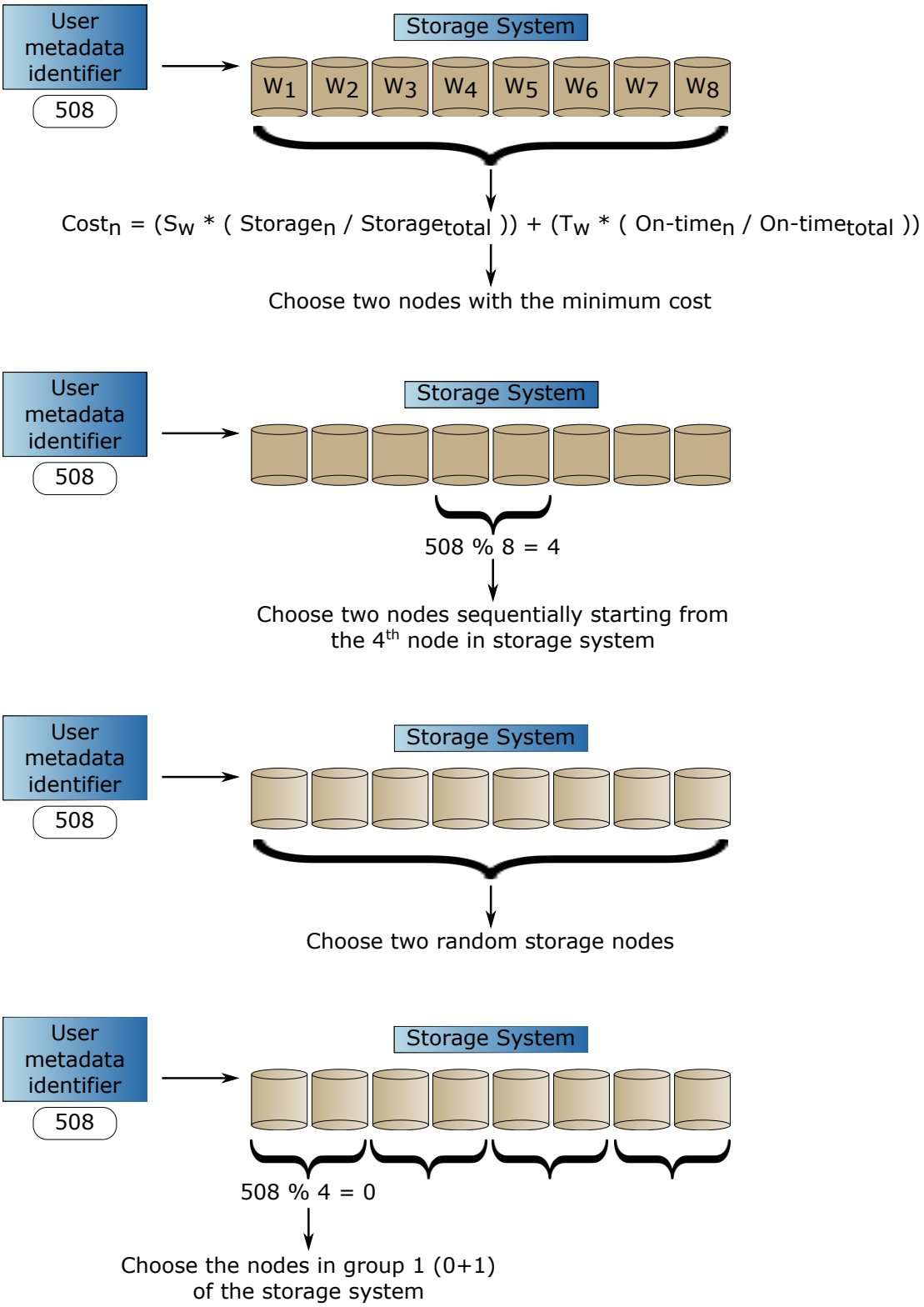


FIGURE 3.2.1: Examples of the node allocation techniques in the Fixed scheme

however, can cause capacity issues as the selected nodes may not have enough storage space and also incurs higher utilization and thus, higher failure rates. The balancing technique enables system administrators to specify weights for *storage space balancing* and *on-time balancing* to indicate their importance and allocates storage nodes for users by adhering to these weights. A *cost* metric is calculated for every storage node in the system by multiplying storage space and on-time weights with the normalized storage space and on-time on that node respectively. In order to normalize storage space and on-time values, they are divided by the total amount of storage and on-time on all nodes respectively. In the end, storage nodes with lower costs are preferred by the balancing technique with the eventual goal of uniformly distributing storage space usage and on-time across storage nodes. As mentioned previously, we assume the storage space usage of a user is distributed evenly across the nodes allocated for that user and a job running on multiple nodes concurrently takes the same amount of time to complete on each node.

Assuming, N is the total number of nodes in the cloud storage system and M is the number of storage nodes allocated for each user, the balancing technique works as shown in Algorithm 1.

The storage space and on-time weights should add up to 1. As an example, if it is equally important to balance storage space and on-time for a system administrator, then $S_W = T_W = 0.5$. However, if it is more important to balance storage space, then $S_W = 1$ and $T_W = 0$.

Algorithm 1 Balancing technique

1: $N \Leftarrow$ Total number of storage nodes
2: $M \Leftarrow$ Number of storage nodes to allocate for the user
3: $ChosenNodes[\,] \Leftarrow$ Nodes that will be allocated for the user
4: $S_{total} \Leftarrow$ Total amount of data stored on all nodes
5: $T_{total} \Leftarrow$ Total duration of time all nodes stayed on
6: $S_i \Leftarrow$ Amount of data stored on node i
7: $T_i \Leftarrow$ Duration of time node i stayed on
8: $S_W \Leftarrow$ Importance of balancing storage space
9: $T_W \Leftarrow$ Importance of balancing time
10: $Costs[\,] \Leftarrow$ Costs of nodes i
11: **for** $i = 1$ to N **do**
12: $\quad Costs[i] \Leftarrow (S_W * (S_i/S_{total}) + (T_W * (T_i/T_{total}))$
13: **end for**
14: **for** $i = 1$ to M **do**
15: $\quad ChosenNodes[i] \Leftarrow min(Costs[\,])$
16: $\quad$ Remove $min(Costs[\,])$ from $Costs[\,]$
17: **end for**

Sequential Technique

The sequential technique uses an approach similar to consistent hashing [74]. The approach starts by calculating an $Offset$ value for a given user with metadata information represented by I. As described earlier, this metadata information can be a user id or user home directory, basically any metadata information that can be hashed to an integer.

Given this hash value, one can then sequentially allocate storage nodes for the user starting from the storage node with an identifier equal to $Offset$. The M nodes following the storage node with identifier equal to $Offset$ are allocated for a user in the sequential technique. A summary of the sequential technique is shown in Algorithm 2.

Note that in Algorithm 2 on line 6, we are taking the modulus of node identifiers

Algorithm 2 Sequential technique

1: $N \Leftarrow$ Total number of storage nodes
2: $M \Leftarrow$ Number of storage nodes to allocate for the user
3: $I \Leftarrow$ Metadata information of the user
4: $Offset \Leftarrow$ Identifier of storage node from which allocation will start
5: $ChosenNodes[\,] \Leftarrow$ Nodes that will be allocated for the user
6: $Offset \Leftarrow I \ mod \ N$
7: **for** $i = 1$ to M **do**
8: $\quad ChosenNodes[i] \Leftarrow (Offset \ + \ i) \ mod \ M$
9: **end for**

in order to make sure they are smaller than the total number of nodes in the cloud storage system, N.

Random Technique

The random technique uses a random number generator that chooses an identifier of a storage node in the system. The random number generator is called M times to generate M different identifiers. These identifiers represent the storage nodes that will be allocated for a user. The random function is seeded with the user's arrival time to the storage system, giving enough randomness. Still, if it produces an identifier twice, it is called until all M identifiers in the subset are distinct. Assuming, N is the total number of nodes in the cloud storage system and M is the number of storage nodes allocated for each user, the random technique works as shown in Algorithm 3.

Grouping Technique

The grouping technique is similar to the sequential technique described in Section 3.2.1. Compared to the sequential technique, the grouping technique has fewer starting points (*offsets*) to choose.

Algorithm 3 Random technique

1: $N \Leftarrow Total\ number\ of\ storage\ nodes$
2: $M \Leftarrow Number\ of\ storage\ nodes\ to\ allocate\ for\ the\ user$
3: $ChosenNodes[\,] \Leftarrow Nodes\ that\ will\ be\ allocated\ for\ the\ user$
4: **for** $i = 1$ to M **do**
5: **while** $j \in ChosenNodes[\,]$ **do**
6: $j \Leftarrow rand()$
7: **end while**
8: $ChosenNodes[i] \Leftarrow j$
9: **end for**

First, node groups of size M are created where M sequential storage nodes together form a group. Here, for ease of explanation, we assume that total number of nodes, N, is a multiple of the number of storage nodes allocated for each user, M. For systems where this is not true, groups may be overlapped. At this point, we assume $G = N / M$ groups are created in the systems, where G denotes the number of groups created. Then, similar to the sequential technique, a group is chosen using the given user's metadata information represented by I and the storage nodes in that group are allocated for that user. As described earlier, this metadata information can be user id or user home directory, basically any metadata information that can be hashed to an integer. The grouping technique works as shown in Algorithm 4.

Algorithm 4 Grouping technique

1: $N \Leftarrow$ Total number of storage nodes
2: $M \Leftarrow$ Number of storage nodes to allocate for the user
3: $G \Leftarrow$ Number of storage node groups
4: $I \Leftarrow$ Metadata information of the user $\Leftarrow N / M$
5: $Offset \Leftarrow$ Identifier of node group which will be allocated for the user
6: $ChosenNodes[\,] \Leftarrow$ Nodes that will be allocated for the user
7: $Offset \Leftarrow I\ mod\ G$
8: **for** $i = 1$ to M **do**
9: $ChosenNodes[i] \Leftarrow (G * M) + i$
10: **end for**

3.2.2 Dynamic Greedy Scheme

In this method, one of the four node allocation techniques described in Section 3.2.1 (balancing, sequential, random or grouping) is initially chosen by the cloud system and that technique is re-evaluated against other techniques at certain times (called as *evaluation points*) over a recent period (called as *control period*), in terms of energy efficiency and/or load balancing. If there is a better technique in terms of energy efficiency and/or load balancing and if the cost of switching to that technique is less than the energy to be saved by switching to that technique, then the current technique is changed. The *Dynamic Greedy Scheme* is described in Algorithm 5.

In this scheme, we use coefficient or variation (CV) amongst the nodes as a proxy for storage space and on-time balancing. In other words, a high CV indicates that the storage space or on-time is not well balanced. The *Dynamic Greedy Scheme* first multiplies the normalized CVs of storage space and on-time ($Svar$ and $Tvar$ in Algorithm 5) with their corresponding importance weights (S_W and T_W). In order to normalize the CVs of storage space and on-time, they are divided by the maximum possible CV of storage space ($MaxSvar$) and on-time ($MaxTvar$) respectively. The maximum CV of storage space and on-time occurs when all requests are served by a single storage node and this sets an upper limit on the CV value. Then the multiplication of the normalized energy cost ($ECost$) with the energy consumption weight (E_W) is added to this product. Energy cost includes both the energy consumption due to running jobs according to a technique ($JobCost$) and the energy consumption due to transferring user data while switching to that technique ($TrnCost$). Energy cost ($ECost$) is divided by the sum of maximum cost of jobs ($MaxJobCost$) and the maximum cost of data transfers ($MaxTrnCost$) over the *control period*. Maximum

job cost ($MaxJobCost$) occurs when all nodes are left on all the time. Maximum data transfer cost ($MaxTrnCost$) occurs when all data existing in the storage system is moved.

A cloud system administrator can specify different values for storage space, on-time and energy consumption weights (S_W, T_W and E_W). This enables a fine-grained control over load-balancing and energy consumption in a cloud storage system. As discussed in Section 3.2.1, the storage space and on-time weights add up to 1. The energy consumption weight is independent of the other two, meaning that, a cloud system administrator might prefer to save energy regardless of load-balancing decisions.

The parameters of a technique evaluation can be summarized as follows:

- **The frequency of the technique evaluations**: The system administrator might decide to evaluate the current node allocation technique of the system against other techniques every couple of hours, days etc.

- **Jobs to evaluate**: Ideally, all jobs that have been submitted to the storage system so far should be evaluated to compare node allocation techniques with each other. However, this might prove to be costly due to the potentially large number of jobs submitted before an *evaluation point*. Therefore, the system administrator specifies a *control period* that specifies the time period from which the jobs are used to evaluate the node allocation techniques at every *evaluation point*. The *control period* again can be couple of hours, days etc., but it has to be less than the frequency of technique evaluations to make sure that evaluations at different moments do not interfere with each other.

- **What is important while comparing node allocation techniques**: Cloud

Algorithm 5 Dynamic Greedy scheme

1: $P \Leftarrow$ Number of available node allocation techniques
2: $S_W \Leftarrow$ Importance of balancing storage space
3: $T_W \Leftarrow$ Importance of balancing time
4: $E_W \Leftarrow$ Importance of reducing energy consumption
5: $Svar_i \Leftarrow$ CV of storage space across all nodes for technique i
6: $MaxSvar \Leftarrow$ Maximum possible CV of storage space across all nodes
7: $Tvar_i \Leftarrow$ CV of on-time across all nodes for technique i
8: $MaxTvar \Leftarrow$ Maximum possible CV of on-time across all nodes
9: $JobCost_i \Leftarrow$ Cost of all jobs in the control period for technique i
10: $TrnCost_i \Leftarrow$ Cost of data transfers while switching to technique i
11: $ECost_i \Leftarrow JobCost_i + TrnCost_i$
12: $MaxJobCost \Leftarrow$ Maximum cost of all jobs over the control period
13: $MaxTrnCost \Leftarrow$ Maximum cost of all data transfers over the control period
14: $TotalCost_i \Leftarrow$ Total cost of technique i at an evaluation point
15: $MinCost \Leftarrow$ Minimum technique cost observed so far
16: $NewTechq \Leftarrow$ Technique that will be in affect after the evaluation
17: $MinCost \Leftarrow 0$
18: $NewTechq \Leftarrow 0$
19: **for** $i = 1$ to P **do**
20: $TotalCost_i \Leftarrow (S_W * (Svar_i \ / \ MaxSvar)) + (T_W * (Tvar_i \ / \ MaxTvar)) + (E_W * (ECost_i \ / \ (MaxJobCost + MaxTrnCost)))$
21: **if** $MinCost == 0$ **or** $TotalCost_i < MinCost$ **then**
22: $NewTechq \Leftarrow i$
23: $MinCost \Leftarrow TotalCost_i$
24: **end if**
25: **end for**

system administrators might be concerned with different aspects of the cloud storage system. As an example, different values can be specified for storage space, on-time and energy consumption weights (S_W, T_W and E_W)

Figure 3.2.2 gives a numerical example of the *Dynamic Greedy Scheme*. A cloud storage system administrator can evaluate node allocation techniques (balancing, sequential, random, grouping) every 4 days by looking at the jobs submitted in the last 6 hours and by trying to save as much energy as possible while trying to balance the storage space only across the nodes. Our approach in this sense is similar to the studies that try to predict idle periods or node reservation lengths [98, 109], but the way we are predicting the future is much simpler compared to those studies. We simply look at the jobs going back to a specified duration of time (a.k.a. *control period*). However, existing studies instrument and monitor cloud systems with complex mechanisms to collect feedback and hints.

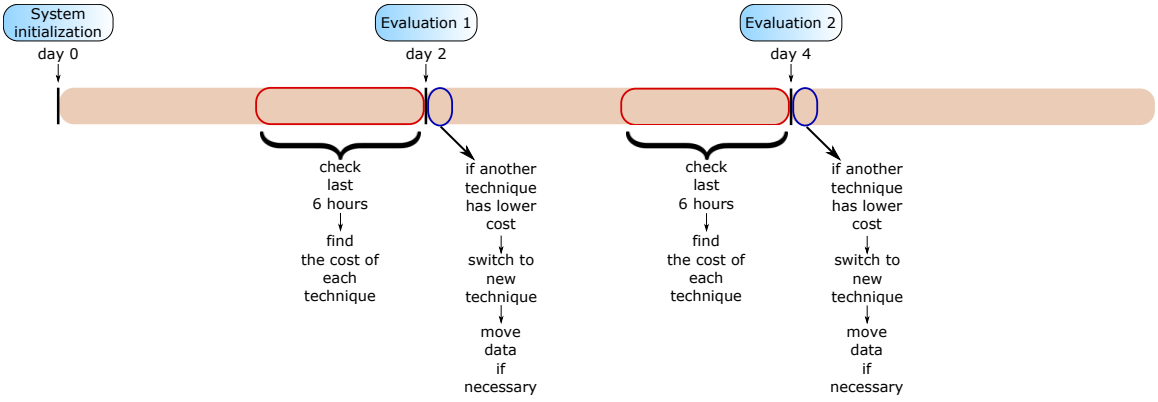


FIGURE 3.2.2: Example of the Dynamic Greedy scheme (evaluations done at every 2 days, last 6 hours are checked only)

3.2.3 Correlation-based Scheme

This scheme is similar to the *Dynamic Greedy Scheme* described in Section 3.2.2, in the sense that the node allocations are evaluated periodically at *evaluation points* over a recent period (called as *control period*). However, rather than switching between node allocation techniques (balancing, sequential, random, grouping) at *evaluation points*, *Correlation-based Scheme* starts with one of these techniques (can be chosen randomly) and keeps using that technique unless it is manually changed. *Correlation-based Scheme* monitors user activities over the recent *control period* and allocates the same subset of cloud storage nodes to the users who tend to use the system in similar fashion. Here, we define the similarity between users as concurrent usage of the cloud storage system, meaning that if two or more users use the cloud storage system for longer than a threshold (called as *similarity threshold* in Algorithm 6) cumulatively, then same subset of storage nodes are allocated for them. We need user access times as the metadata information to find concurrent user accesses, but other metadata (i.e. number of accesses, amount of data stored) can be used as well to assess similarity between users. Figure 3.2.3 gives a numerical example of the *Correlation-based Scheme*.

The *similarity threshold* sets a lower limit to define the concurrent usage of the storage system. As an example, if the *similarity threshold* is 4 hours, two or more users using the cloud storage systems for more than 4 hours cumulatively are going to be assigned the same storage nodes. Here, it is important to point out that the *Correlation-based Scheme* evaluates concurrent user activities cumulatively, meaning that, if two or more users access the system concurrently for 3 hours in the morning and then for 1 hour in the afternoon, those users will be considered as using the stor-

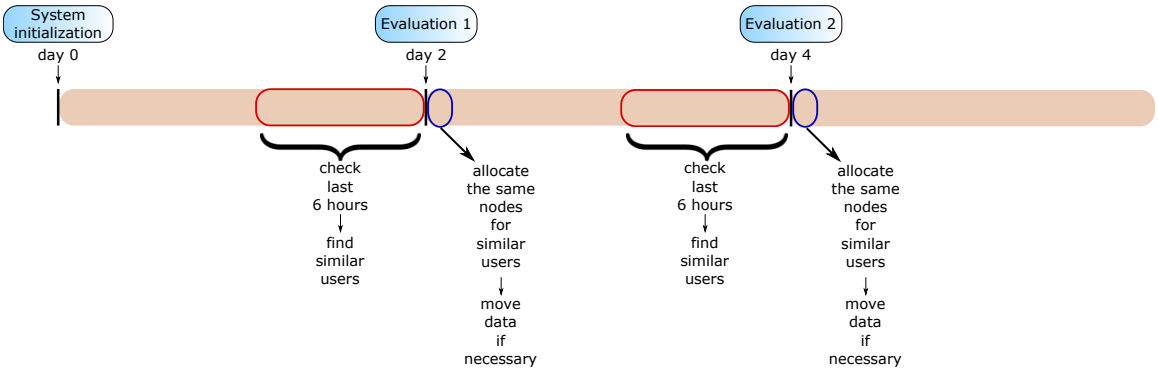


FIGURE 3.2.3: Example of the Correlation-based scheme (evaluations done at every 2 days, last 6 hours are checked only)

age system in a similar fashion and they will be assigned the same subset of cloud storage nodes. The *Correlation-based Scheme* is described in Algorithm 6.

Algorithm 6 Correlation-based scheme

1: $T_{similar} \Leftarrow Similarity\ threshold$
2: $R \Leftarrow Number\ of\ similarities\ between\ users\ in\ the\ most\ recent\ control\ period$
3: $Similarities[\,] \Leftarrow Array\ storing\ duration\ of\ user\ similarities$
4: **for** $i = 1$ to R **do**
5: **if** $Similarities[i] > T_{similar}$ **then**
6: Allocate the same storage nodes for users in this similarity
7: **end if**
8: **end for**

3.3 Mathematical Model of the Problem

In this section, we present a mathematical model to estimate the outcome of the proposed methods with varying parameters. We are particularly interested in estimating the energy consumption, latency per access and load balance as the outcomes of the system. The methods we proposed distribute jobs in a workload across all of the storage nodes, eventually creating time interval series on each storage node. The

main driving parameter of the mathematical model is these series of time intervals. We consider three different time intervals; *interjobs, interarrivals* and *job lengths*. Interjob is the interval between the completion of a job and the start of the next job. Interarrival is the interval between the start of a job and the start of the next job. Job length is the interval between the start and completion of a job. We collect these time series data for each workload and storage node and fit them to probability distribution models using the *allfitdist* function [11] of MATLAB in order to estimate the energy consumption, latency per access and load balance.

In the proposed mathematical model, time series can be fitted to different probability distribution models depending on the workload and the number of jobs assigned to a storage node. As each workload can be fitted to different probability distribution models on each storage node, we use $F(t)$ to represent the cumulative distribution function (CDF) at this point for ease of presentation. By definition of the cumulative distribution function, $F_X(t)$ represents the probability that the random variable X, i.e. the interarrival time between jobs, is smaller than or equal to t. $F(t)$ is replaced with the actual CDF of each workload in Section 3.5.6.

3.3.1 Energy Consumption

We first estimate the energy consumption of a storage node. The average job length on a storage node without taking any latencies into account can be calculated by looking at the jobs on a storage node and taking their average as shown in Equation (3.3.1), where N_J is the number of jobs on a storage node, J_{avg} is the average job length without latencies and $J(k)$ is the length of an individual job.

$$J_{avg} = \frac{\sum_{k=1}^{N_J} J(k)}{N_J} \tag{3.3.1}$$

In order to estimate the impact of job latencies, we consider four different cases for the interjob times:

- The interjob time is greater than the sum of the inactivity threshold (T_I) and the startup time (T_S). In this case, even if the job finishing before the interjob period experiences latency, the interjob time will still be greater than the inactivity threshold and therefore will cause the storage node to be turned off. In this case, the next job assigned to this storage node will experience latency equal to the startup time, since it will have to wait for the storage node to be turned on. The average latency impact is the probability that the interjob time is larger than $(T_I + T_S)$ multiplied by $(T_I + T_S)$, or $(1 - F_{T_I+T_S}(t)) * (T_I + T_S))$.

- The interjob time is greater than the inactivity threshold, but smaller than or equal to the sum of the inactivity threshold and startup time. In this case, if the job finishing before the interjob period experiences latency, the interjob time might become smaller than the inactivity threshold. This will cause the storage node not to be turned off at all and therefore to miss the opportunity to reduce energy consumption. The average latency impact is the probability that the interjob time is between T_I and $T_I + T_S$ multiplied by the interjob time, or $\int_{T_I}^{T_I+T_S} tF'(t)dt$.

- The interjob time is smaller than or equal to the inactivity threshold, but greater than the startup time. In this case, even if the job finishing before the interjob

period experiences latency, the storage system will stay on until the next job arrives. The average latency impact is the probability that the interjob time is between T_S and T_I multiplied by the interjob time, or $\int_{T_S}^{T_I} tF'(t)dt$.

- The interjob time is smaller than or equal to the startup time. In this case, if the job finishing before the interjob period experiences latency, it might overlap with the next job assigned to the storage node, as the latency it experiences (startup time) will be greater than the interjob time. If there is an overlap between jobs, that means we are reducing energy consumption. The average latency impact is the probability that the interjob time is less than T_S multiplied by the interjob time, or $-\int_0^{T_S} tF'(t)dt$.

Putting these impacts together, we can estimate the average job length with latencies as shown in Equation (3.3.2).

$$\begin{aligned} J_{avg}^{'} = J_{avg} \quad & + (1 - F_{T_I+T_S}(t)) * (T_I + T_S) \\ & + \int_{T_I}^{T_I+T_S} tF'(t)dt \\ & + \int_{T_S}^{T_I} tF'(t)dt \\ & - \int_{0}^{T_S} tF'(t)dt \end{aligned} \tag{3.3.2}$$

The total time (T) a storage node stays on (on-time) can be estimated by using the average job length with latencies ($J_{avg}^{'}$) and the number of jobs on a storage node (N_J). Then the total time can be multiplied with power of a storage node to find the energy consumption, as shown in Equation (3.3.3).

$$\begin{aligned} Energy\ \ consumption\ \ of\ \ a\ \ node &= Power\ \ *\ \ T, \\ where\ \ T\ \ &= N_J\ \ *\ \ J^{'}_{avg} \end{aligned} \tag{3.3.3}$$

3.3.2 Load Balancing

Equation (3.3.3) shows the total time a storage node stays on; where N_J is the number of jobs on that storage node and $J^{'}_{avg}$ is the average job length with latencies. While balancing the on-time across all storage nodes, we consider the coefficient of variation (CV) of on-time, which is denoted by CV_T. Therefore, we can formulate balancing the on-time across storage nodes with Equation (3.3.4).

$$CV_T = \frac{\sqrt[2]{\frac{\sum_{i=1}^{N}(T_i - T_{mean})^2}{N}}}{T_{mean}}; \tag{3.3.4}$$

where T_{mean} is found as follows, with N being the number of storage nodes and T_i being the total time storage node i stays on;

$$T_{mean} = \frac{\sum_{i=1}^{N} T_i}{N}; \tag{3.3.5}$$

Balancing storage space across storage nodes is represented with the same model shown in Equation (3.3.4) and (3.3.5). The only difference will be using storage space instead of on-time. We model balancing the storage space across storage nodes with

Equation (3.3.6). CV_S denotes the coefficient of variation of storage space across all nodes.

$$CV_S = \frac{\sqrt[2]{\frac{\sum_{i=1}^{N}(S_i - S_{mean})^2}{N}}}{S_{mean}}; \tag{3.3.6}$$

where S_{mean} is found as follows, with N being the number of storage nodes and S_i being the used storage space on node i;

$$S_{mean} = \frac{\sum_{i=1}^{N} S_i}{N}; \tag{3.3.7}$$

3.3.3 Latency per Access

As discussed before, any job that is assigned to a turned-off storage node needs to wait for that node to startup, an operation that will take time equal to T_S. However, any subsequent job that is assigned to the same storage node before that node fully starts up will also experience latency. The latency experienced by these subsequent jobs will be smaller than T_S; therefore, we need to estimate the effective latency of jobs arriving to a storage node while that node is being started. In order to estimate the number of jobs arriving to a storage node in a certain period of time and to find out the latency experienced by each, we need to examine the interarrival times of a workload. After fitting interarrivals to probability distribution functions, we can estimate how many jobs will arrive to a storage node when it is being started.

The first job that triggers the start of a turned-off storage node will always experience a latency that is equal to T_S. Any subsequent jobs that arrive to that storage node before it is fully turned on will experience a smaller amount of latency. Therefore, assuming a new job arrives every second, we can formulate the effective latency of a storage node as shown in Equation (3.3.8).

$$t_{eff} = T_S + \int_0^{T_S} (T_S - t) * F'(t)dt \tag{3.3.8}$$

In order to estimate latency per access, we need to take the average of effective latency values on all storage nodes. This is shown in Equation (3.3.9), where $t_{eff}(i)$ is the effective latency on storage node i, $N_I(i)$ is the number of interjob periods on storage node i, $P_I(i)$ is the probability of an interjob period being greater than the inactivity threshold (therefore causing the next job to experience latency) and N is the number of storage nodes in the system.

$$\begin{gathered} Latency \;\; per \;\; access = \frac{\sum_{i=1}^{N} t_{eff}(i) * N_I(i) * P_I(i)}{N}, \\ where \;\; P_I \;\; = \;\; 1 \;\; - \;\; F_{T_I}(t) \;\; on \;\; a \;\; storage \;\; node \end{gathered} \tag{3.3.9}$$

3.4 Theoretical Optimality Analysis

In this section we give theoretical analysis of the proposed methods in order to learn the approximation factor of each to the best possible solution in terms of energy consumption or load balancing. We calculate the approximation factor (C) of each method by comparing its energy consumption or load balancing value with the optimal value. C is the upper bound on the worst case performance of our methods; such that, when the optimal solution is multiplied by C, it is guaranteed to be equal to or greater than solutions provided by our methods. C can be formulated as shown in (3.4.1).

$$P_{proposed} <= C * P_{optimum} \tag{3.4.1}$$

where C is the approximation factor, $P_{proposed}$ is the solution provided by one of our methods and $P_{optimum}$ is the optimal solution.

3.4.1 Load Balancing

Storage Space

Our proposed methods try to balance storage space across storage nodes by trying to distribute new or transferred users as evenly as possible. We assume that the optimal solution knows the storage space usage of each user beforehand; so that, it makes the best decision in terms of storage space balancing when a user comes to the system for the first time or when its storage is transferred between the storage nodes.

We assume that there are N cloud storage nodes, where M of them are allocated

per user and there are K total users using the system. $S_{node}(i)$ represents the total eventual storage space usage on storage node i, $S_{node}(i)(t)$ represents the total storage space usage on storage node i at time t and $S_{user}(i)$ represents the storage space usage of user i. The optimal solution will distribute users as evenly as possible, such that, the makespan of the cloud storage system (maximum storage space usage on a storage node in the entire system) is minimized [51].

There are two boundaries for the optimal solution. It will either be able to allocate all N storage nodes or only one storage node per user. We denote these two cases as *Case1* and *Case2* and use the maximum of these two as the optimal solution for storage space balancing $P_{optimum}$ as shown in Equation (3.4.4).

- *Case1* The optimal solution allocates all N storage nodes per user. In this case, the storage space is perfectly distributed and the makespan will be found as shown in Equation (3.4.2).

$$P_{optimum1} = \frac{\sum_{i=1}^{K} S_{user}(i)}{N} \tag{3.4.2}$$

- *Case2* The optimal solution allocates only a single storage node per user. In this case, the makespan of the storage system will be found as shown in Equation (3.4.3).

$$P_{optimum2} = \max S_{user}(i), \quad where \;\; 1 <= i <= K \tag{3.4.3}$$

$$P_{optimum} = \max P_{optimum1}, P_{optimum2} \tag{3.4.4}$$

When a new or transferred user comes to the cloud storage system, all but two of our proposed methods might allocate the storage nodes with maximum storage space usage to this user. We denote the solution of these methods as $P_{proposed\_worst}$. The two cases that will try to achieve the best possible storage space distribution are balancing technique with storage space balancing weight (S_W) set to one and Dynamic Greedy scheme that has balancing technique as the initial technique with storage space balancing weight again set to one and energy consumption weight (E_W) set to zero. Similarly, we denote the solution of these two methods as $P_{proposed\_best}$.

We first analyze the approximation factor of $P_{proposed\_best}$. In this case, two methods mentioned in the previous paragraph allocate M storage nodes with the minimum storage space usage to a user. As it will not affect our theoretical analysis, we assume $M = 1$. We also assume that storage node f has the biggest storage space usage at time T_1 as in Equation (3.4.5), after it was allocated for user u that has a storage space usage of $S_{user}(u)$.

$$S_{node}(f)(T_1) = \max S_{node}(i)(T_1), \quad where \;\; 1 <= i <= N \tag{3.4.5}$$

This means that at time T_2 ($T_2 < T_1$), when the decision to allocate storage node f for user u was made, node f had the minimum storage space usage in the system as in Equation (3.4.6).

$$N * S_{node}(f)(T_2) <= \sum_{i=1}^{N} S_{node}(i)(T_2) \tag{3.4.6}$$

As the total storage space in the system at time T_2 will be less than the total eventual storage space usage, we can infer Equation (3.4.7).

$$\sum_{i=1}^{N} S_{node}(i)(T_2) < \sum_{i=1}^{N} S_{node}(i) \tag{3.4.7}$$

Another observation we can make is that the total eventual storage space usage on the system will be equal to the sum of individual storage space usage of each user as shown in Equation (3.4.8).

$$\sum_{i=1}^{N} S_{node}(i) = \sum_{i=1}^{K} S_{user}(i) \tag{3.4.8}$$

And from Equation (3.4.2), (3.4.3) and (3.4.4), we can infer that;

$$\sum_{i=1}^{K} S_{user}(i) <= N * P_{optimum} \tag{3.4.9}$$

Therefore, combining Equation (3.4.6), (3.4.7), (3.4.8) and (3.4.9), we can obtain;

$$S_{node}(f)(T_2) < P_{optimum} \tag{3.4.10}$$

We now know that storage node f has the maximum amount of storage space used in the system at time T_1 and the minimum amount of storage space used in the system at time T_2 (before it was allocated for user u that has a storage space usage of $S_{user}(u)$). Thus, we can infer the equality in Equation (3.4.11).

$$S_{node}(f)(T_1) = S_{node}(f)(T_2) + S_{user}(u) \tag{3.4.11}$$

We can also observe the following from Equation (3.4.2), (3.4.3) and (3.4.4).

$$S_{user}(u) <= P_{optimum} \tag{3.4.12}$$

Therefore, combining Equation (3.4.10), (3.4.11) and (3.4.12) we can see that two of the proposed methods (balancing technique with $S_W = 1$ and Dynamic Greedy scheme that has balancing technique as the initial technique $S_W = 1$ and $E_W = 0$) have an approximation factor of 2 in the best case.

$$P_{proposed\_best} = S_{node}(f)(T_1) < 2 * P_{optimum} \tag{3.4.13}$$

We now analyze the approximation factor of other methods, $P_{proposed\_worst}$. These methods might allocate M storage nodes with the maximum storage space usage to a user in the worst case. We again assume $M = 1$ as it will not change the theoretical analysis. The theoretical analysis of $P_{proposed\_worst}$ will be similar to that of $P_{proposed\_best}$. In fact, the only difference will be to Equation (3.4.6), as the storage node

with the maximum storage usage at time T_2 will definitely have storage space usage less than or equal to the total storage space of all nodes as shown in Equation (3.4.14).

$$S_{node}(f)(T_2) <= \sum_{i=1}^{N} S_{node}(i)(T_2) \tag{3.4.14}$$

This will change Equation (3.4.10) as follows;

$$S_{node}(f)(T_2) < N * P_{optimum} \tag{3.4.15}$$

Therefore, we can see that the proposed methods have an approximation ratio of $N+1$ in the worst case, where N is the total number of storage nodes in the system.

$$P_{proposed\_worst} = S_{node}(f)(T_1) < (N+1) * P_{optimum} \tag{3.4.16}$$

On-Time

Theoretical analysis of our proposed methods in terms of balancing on-time across storage nodes will be exactly the same as the theoretical analysis of balancing storage space; except for the following;

- On-time information will be used instead of storage space.
- Techniques producing $P_{proposed\_best}$ and $P_{proposed\_worst}$ will be different.

The balancing technique with on-time balancing weight (T_W) set to one and Dynamic Greedy scheme that has balancing technique as the initial technique with on-time balancing weight set to one and energy weight (E_W) set to zero will produce a 2-approximation $P_{proposed\_best}$ solution, whereas other methods will produce an (N + 1)-approximation $P_{proposed\_worst}$ solution compared to the optimal solution in terms of balancing on-time.

3.4.2 Energy Consumption

We compare the energy consumption of the proposed methods with that of the optimal solution in order to find the approximation factor of each. We assume again there are N storage nodes in the system and M of them are allocated for each user. The theoretical model for the energy consumption is based on *interjob* times that were described earlier in Section 3.3.

In the best case, the storage node that is allocated for a user by our proposed methods will be already on. In this case, there will not be any difference between the optimal solution and the solution provided by our provided methods.

The worst case for any of the methods we propose will be when all of the storage nodes have been turned off due to staying inactive for longer than the inactivity threshold; such that, a job submitted to the M storage nodes at this time will experience latency equal to $T_I + T_S$ regardless of the method we are using. T_I represents the inactivity threshold and T_S represents the startup time of a storage node.

On the other hand, the optimal solution knows when an idle period will begin and end. Therefore, it can turn off a storage node without waiting for the inactivity threshold to elapse. As a result, $P_{proposed}$, $P_{optimum}$ and C can be formulated as shown

in Equation (3.4.17), (3.4.18) and (3.4.19).

$$P_{proposed} = \sum_{i=1}^{M} (T_I + T_S) \tag{3.4.17}$$

$$P_{optimum} = \sum_{i=1}^{M} T_S \tag{3.4.18}$$

$$\begin{aligned} C >= \frac{(T_I + T_S)}{T_S} \\ C >= \frac{(k * T_S + T_S)}{T_S} = k + 1, \quad where \quad k = \frac{T_I}{T_S} \end{aligned} \tag{3.4.19}$$

Consequently, if any of our methods use an inactivity threshold (T_I) value equal to the startup time (T_S), the approximation factor will be 2.

3.5 Results

In this section, we first present experimental simulation results and then validate the mathematical model presented in Section 3.3.

3.5.1 Experimental Setup

In order to quantify the impact of our energy conservation methods, we use various workloads that approximate usage in a cloud storage system. The three workloads come from Google, the University of Connecticut Hornet HPC system and the

GRID5000 platform. These workloads are described in more detail below. The jobs in these workloads are tied to users and each job is assumed to use the storage system equally.

Workloads

Google Trace The Google Trace is a log of jobs that are run in Google's cloud system [134]. This trace has 29 days of data collected from nearly 11,000 machines in May 2011 and it stores detailed information (including storage space used) for each *task*, *job* and *machine event*. We use the data from individual tasks in the evaluations as each job comprises multiple tasks. Each task has a status code indicating the current status of the task - such as submission, failure, re-submission, completion etc. For our test purposes, we only consider tasks that are submitted and completed successfully. The Google trace is a highly intensive workload, consisting of nearly 16 million tasks. We found the number of users in this workload to be 100 (after unhashing the user names).

Hornet Cluster The Hornet Cluster [7] is a high end cluster at the University of Connecticut consisting of 64 nodes where each node has 12 Intel Xeon X5650 Westmere cores, 48 GB of RAM and 500 GB of storage. Information about each job executed on this cluster is recorded in the log files. For each job, the log files provide the identifier of the user who submitted the job, in addition to the submission and completion times. The log file comprises 25 months of data collected between September 2011 and October 2013. While the Hornet cluster is not a cloud system,

it is representative of cluster jobs that may run on a cloud compute system. The Hornet cluster workload includes about 145,000 jobs from 307 users.

GRID5000 Workload GRID5000 is a grid platform located in France comprising of nine separate computing sites with a total of fifteen clusters available. Although a grid may not necessarily be classified as a cloud platform, it has an architecture similar to a cloud system, making GRID5000 a useful workload for our evaluations. GRID5000 workload [23] stores information about jobs submitted between May 2004 and November 2006. There is no record of storage space usage in this workload. Therefore, we randomly generated that for each user. GRID5000 workload consists of nearly one million jobs from 645 users.

Test Parameters

We test each workload with the parameters shown in Table 3.5.1. We assume each storage node consumes 300 Watts of power [39].

3.5.2 Comparison of Energy Savings

Before delving into detailed analysis of each proposed method, we first present a general comparison of the proposed methods in terms of energy consumption as shown in Figure 3.5.1. This figure shows the energy consumption of each workload in the Fixed scheme, Dynamic Greedy scheme, Correlation-based scheme and the stock case, where there is no energy saving mechanism in place (system always on). The total number of storage nodes in the system is 64 and 8 nodes are allocated for each user. Additionally, the initial technique is balancing with both storage space and on-time

Test Parameter	Values
Total number of nodes	64
Number of nodes allocated for each user	8
Allocation methods	Fixed, Dynamic Greedy, Correlation-based
Low-energy mode	Turn Off
Inactivity threshold	300, 1800, 3600 s
Startup time	30, 60, 90 s
Similarity threshold	3600, 7200, 14400 s
Storage space and on-time balancing weights	0, 0.5, 1
Power consumption per node	300 W
Workloads	Google, Hornet, GRID5000

TABLE 3.5.1: Test parameters

balancing weights (S_W *and* T_W) equal to 0.5 and energy consumption weight (E_W) equal to one. For Dynamic Greedy and Correlation-based schemes evaluations are done every 20 days and the last 24 hours are checked at every evaluation. For the Correlation-based scheme the similarity threshold is one hour.

Figure 3.5.1 shows that, for the Google trace any of the methods would deliver the same amount of energy savings and the percentage of energy savings is 8.4% at its maximum. Google trace is a highly I/O intensive workload and there are not many idle periods in the system. As a result, energy savings are not significant. For Hornet cluster data and GRID5000 workload, we observe that all three workloads save significant amount of energy. For both of these workloads, we see that Dynamic Greedy scheme does not improve on energy consumption that much compared to the Fixed scheme. One of the main objectives of the Dynamic Greedy scheme is to balance load across the storage system. As a result, balancing the load across the system prevents further energy savings. Correlation-based scheme provides slightly better

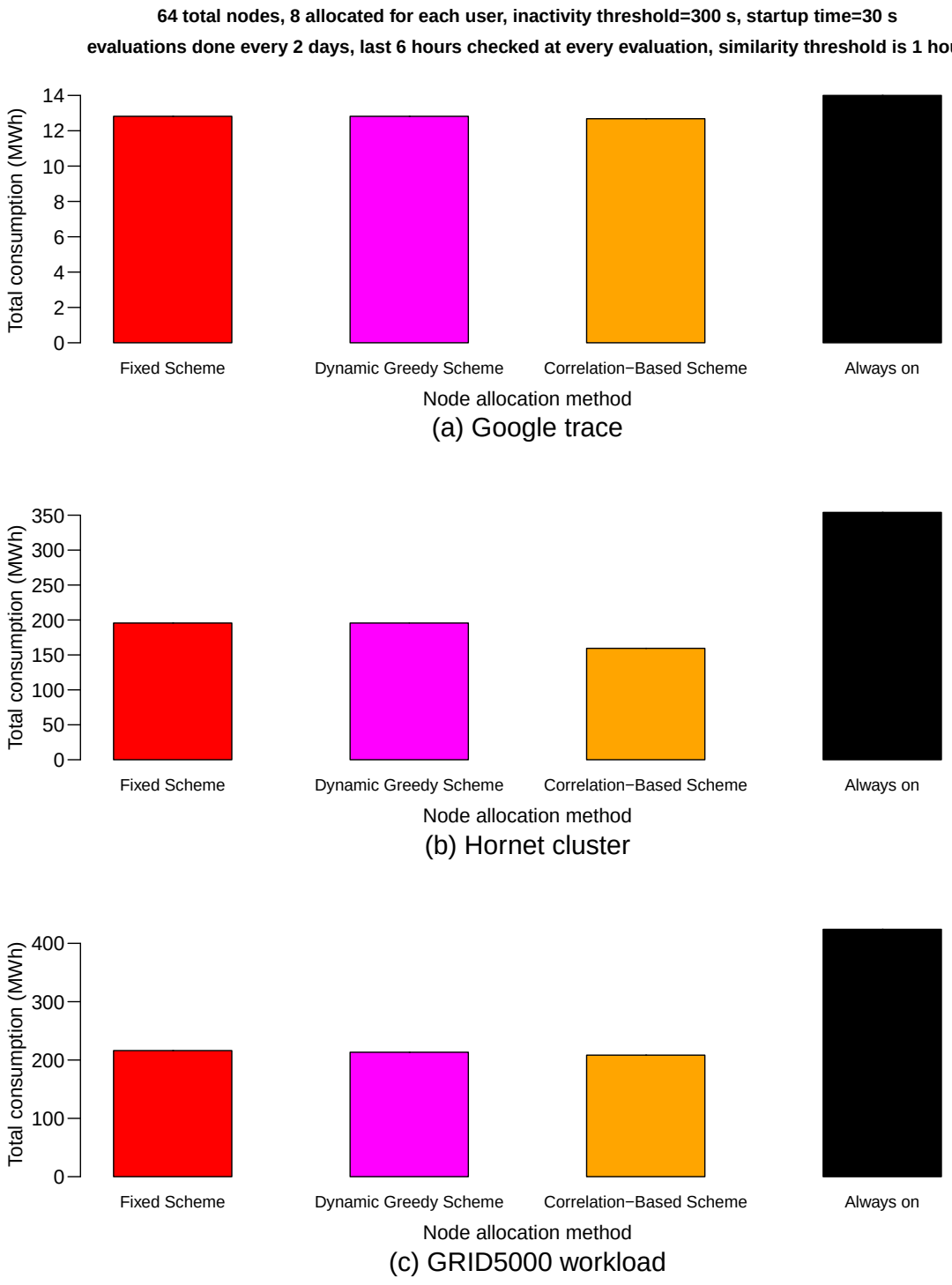


FIGURE 3.5.1: Energy consumption comparison of methods

energy savings for GRID5000 workload and much better energy savings for Hornet cluster data. Hornet cluster is mostly used by researchers working simultaneously on a project which potentially improves the correlation between storage system users. For Hornet cluster data the energy savings are as high as 55% and for GRID5000 workload energy savings are as high as 50.9%. In related studies, Wildani et al. [132] achieved 52% of energy savings; but, compared to the case where the disks spin up for every read operation. Similarly, PDC [104] and MAID [48] conserved up to 30-40% of the energy compared to the case where disks were never powered down; but, these savings were achieved only for low server loads.

3.5.3 Fixed Scheme

We first evaluate the Fixed scheme, where one of the node allocation techniques (balancing, sequential, random or groups) is chosen and kept active unless manually changed. We calculate the total energy consumption, load balance and latency per access with 64 total nodes in the system, where 8 nodes are allocated for each user. Figure 3.5.2 shows how energy consumption, load balance and latency per access change with each node allocation technique. *Random* node allocation technique is executed five times and the average result is reported, as it can yield to a different result at each run. Due to space considerations, we show only the Google trace evaluations of the Fixed scheme, as other workloads have similar results.

As seen in Figure 3.5.2, each node allocation technique saves some amount of energy compared to the stock case where there is no energy saving technique in place (system always on). The percentage of savings is around 8.4% for the Google trace. Google trace is highly intensive and as a result, there are not that many idle periods

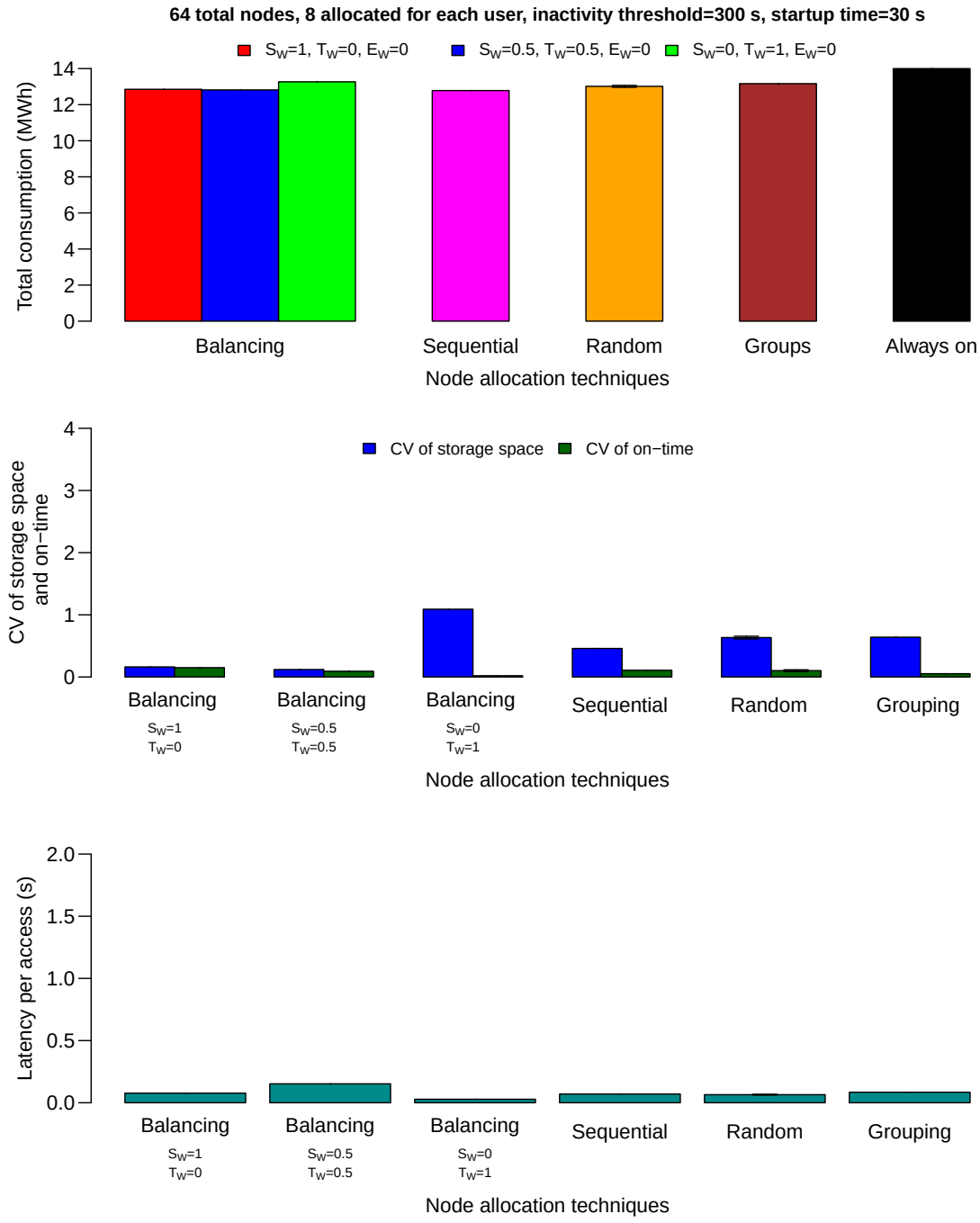


FIGURE 3.5.2: Allocation technique vs energy consumption, load balance and latency per access (Fixed scheme - Google trace)

in this workload. Therefore, energy savings are limited in such a busy workload. The amount of energy savings does not vary much with different techniques. Balancing technique with on-time balancing weight (T_W) set to one consumes slightly more energy compared to other techniques. The primary objective of balancing technique with on-time balancing weight set to one is to balance the on-time of nodes across the storage system. We found from the Google trace that majority of the storage system accesses are in the range of couple hundred seconds. Google trace workload is already balanced initially in terms of node on-time; therefore, balancing technique with on-time balancing weight set to one does not change the node allocations as much as other techniques do, resulting in higher energy consumption.

However, the techniques do differ considerably in terms of balancing storage space and on-time. Second sub-plot in Figure 3.5.2 shows how the coefficient of variation (CV) of storage space and on-time across the storage nodes vary with different node allocation techniques. As shown in the second sub-plot of Figure 3.5.2, the balancing technique with non-zero storage space balancing weight (S_W) has smaller CV of storage space across the storage nodes. Other techniques do not perform that well as their primary objective is not to balance storage space across the storage nodes.

The balancing technique also performs well for balancing on-time of the storage nodes, as shown in the second sub-plot of Figure 3.5.2. Balancing technique with time balancing weight (T_W) set to one has the smallest CV of on-time across the storage nodes as expected.

The third sub-plot in Figure 3.5.2 shows how the latency per access changes with each node allocation technique. As we have mentioned previously, balancing technique with on-time balancing weight set to one does not change the node allocations as much as other techniques do. As a result, that technique has the smallest latency

per access. On the other hand, balancing technique with both storage and on-time balancing weight set to 0.5 has the highest energy savings compared to other techniques; increasing its latency per access. In general, we can see that all techniques have very close latency per access values, usually around 0.1 seconds.

We also investigate how the total energy consumption, load balance and latency per access is affected by the changes in the startup time and inactivity threshold. Figure 3.5.3 shows the total energy consumption, load balance and latency per access measurements while the startup time is varied between 30, 60 and 90 seconds. Similarly, Figure 3.5.4 shows the total energy consumption, load balance and latency per access measurements while the inactivity threshold is varied between 300, 1800 and 3600 seconds. As each technique is affected similarly by the changes in the inactivity threshold and startup time, we only present results for the balancing technique with both storage space and on-time balancing weights (S_W and T_W) equal to 0.5 and energy consumption weight (E_W) equal to zero.

As we can see in both Figure 3.5.3 and Figure 3.5.4, changing the inactivity threshold or the startup time has nearly no effect on the load balance. Changing the startup time does not affect the energy consumption either. Figure 3.5.4 shows that increasing the inactivity threshold causes energy consumption to go up. When the inactivity threshold is higher, it is less likely for a storage node to be turned-off for being idle longer than this threshold. This causes the storage node to stay on for longer and consequently increases energy consumption. Latency per access slightly increases as the startup time is increased. This is expected, since any access that is made to a turned-off node will wait longer for that node to startup again. On the other hand, as the inactivity threshold is increased, latency per access slightly decreases. In that case, the storage nodes are kept on for longer due to higher inactivity threshold;

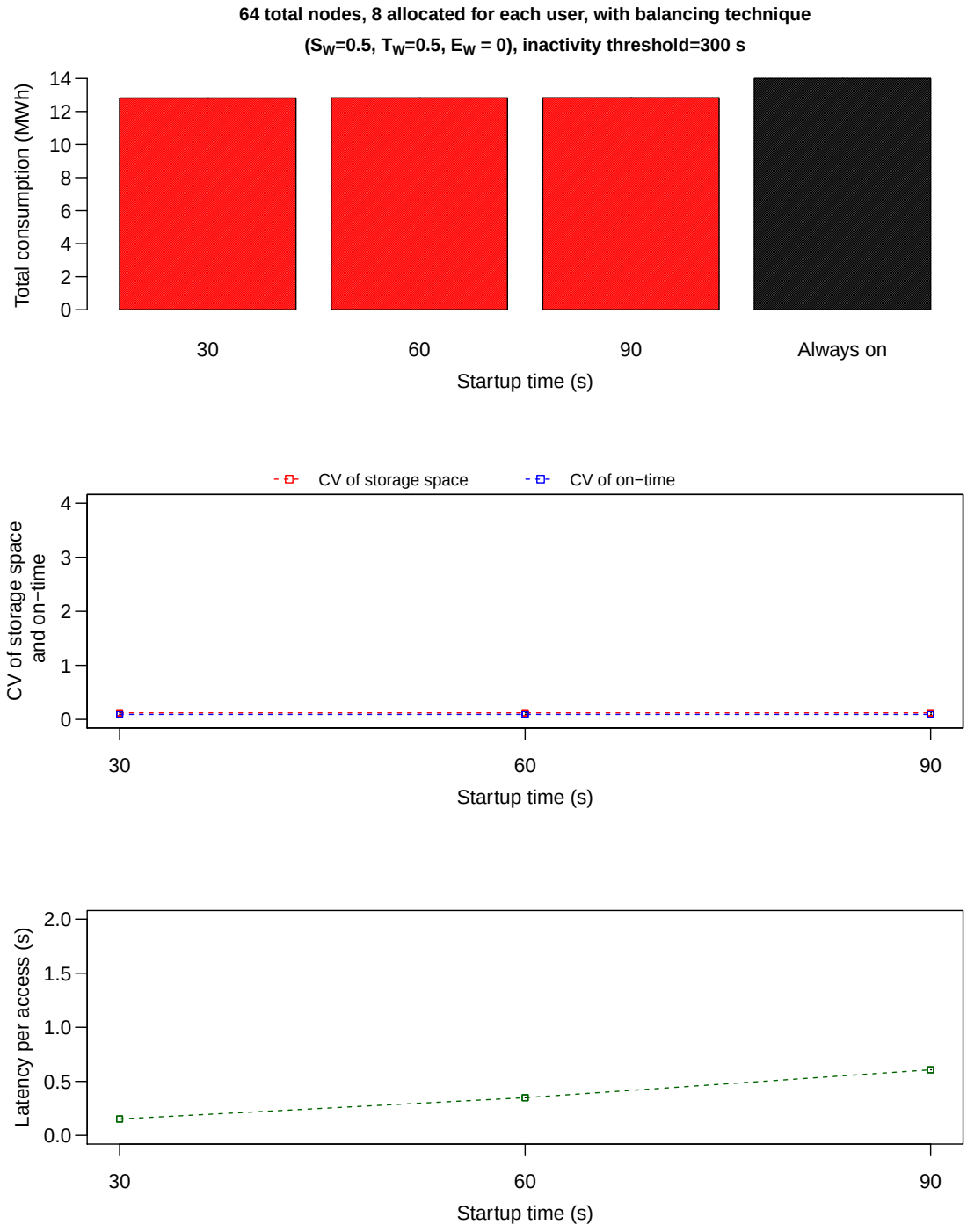


FIGURE 3.5.3: The effect of varying the startup time (Fixed scheme - Google trace)

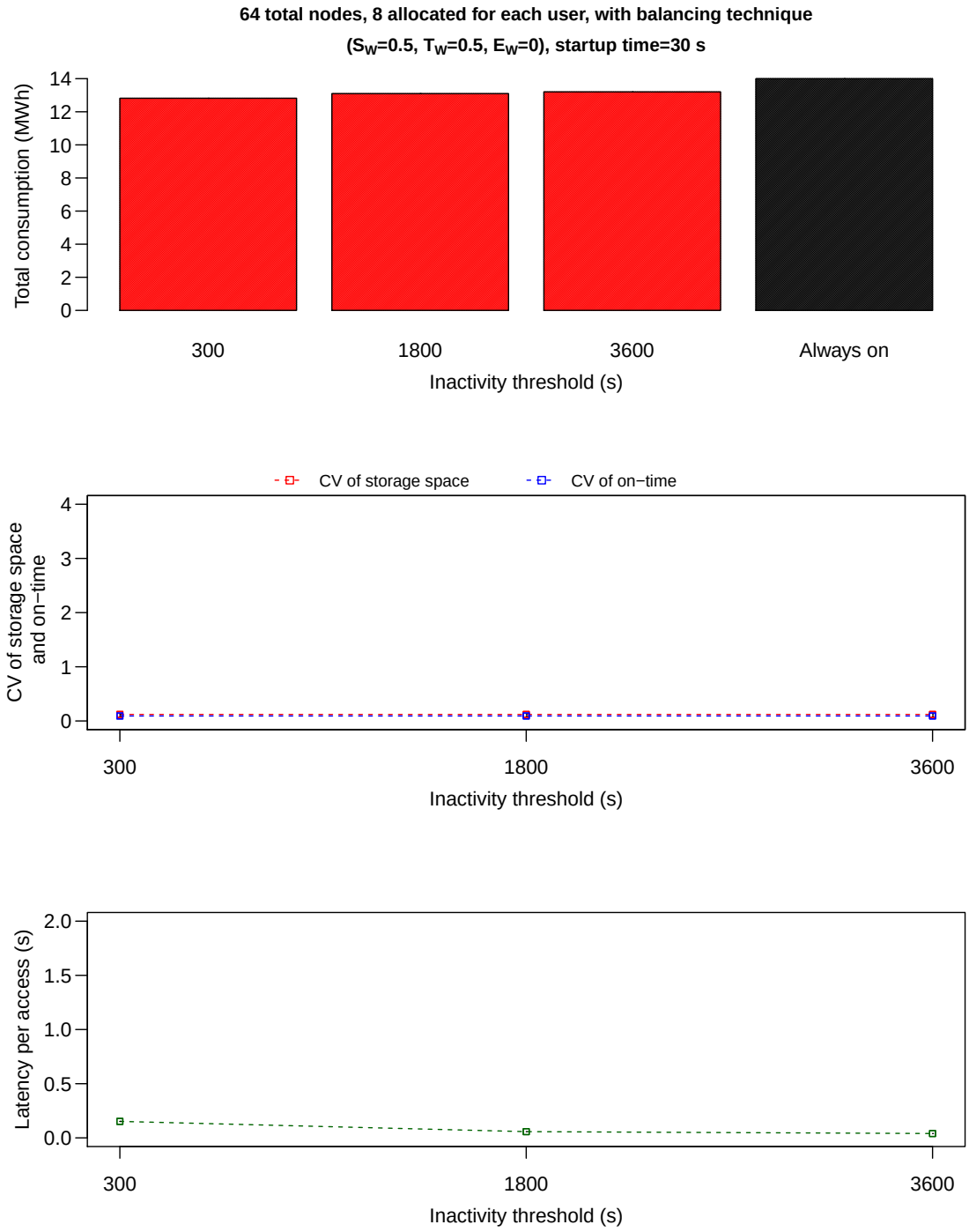


FIGURE 3.5.4: The effect of varying the inactivity threshold (Fixed scheme - Google trace)

which means that less storage system accesses will be made to a turned-off node.

3.5.4 Dynamic Greedy Scheme

In this section, we evaluate the Dynamic Greedy scheme and find out how effective it is in terms of energy consumption, load balance and latency per access. As discussed in Section 3.2.2, *evaluation points* and *control periods* are two important parameters of the Dynamic Greedy scheme. Dynamic Greedy scheme starts with one of the four node allocation techniques we proposed (balancing, sequential, random or grouping) and changes between these techniques at evaluation points, if necessary. We try to understand if the initial technique chosen has any effect on the energy consumption, load balance or latency. We also analyze the effect of varying evaluation points and control periods on the energy consumption, load balance and latency per access.

To start with, we test the Dynamic Greedy scheme with varying storage-space, on-time and energy consumption weights (S_W, T_W and E_W) where the total number of storage nodes in the system is 64 and the number of nodes allocated for each user is 8, as shown in Figure 3.5.5. The evaluations are done every 2 days and the storage accesses in the last 6 hours are checked. Each measurement is the average of five runs. Due to space considerations, we show only the Hornet cluster evaluations of the Dynamic Greedy scheme, as other workloads have similar results.

Figure 3.5.5 shows that the Dynamic Greedy scheme saves a considerable amount of energy regardless of which node allocation technique is chosen initially. The percentage of energy savings are between 43.4% and 52% for varying storage space, on-time and energy weights. Regardless of which technique is initially chosen, we can see that cases where the energy consumption weight (E_W) is set to one consume

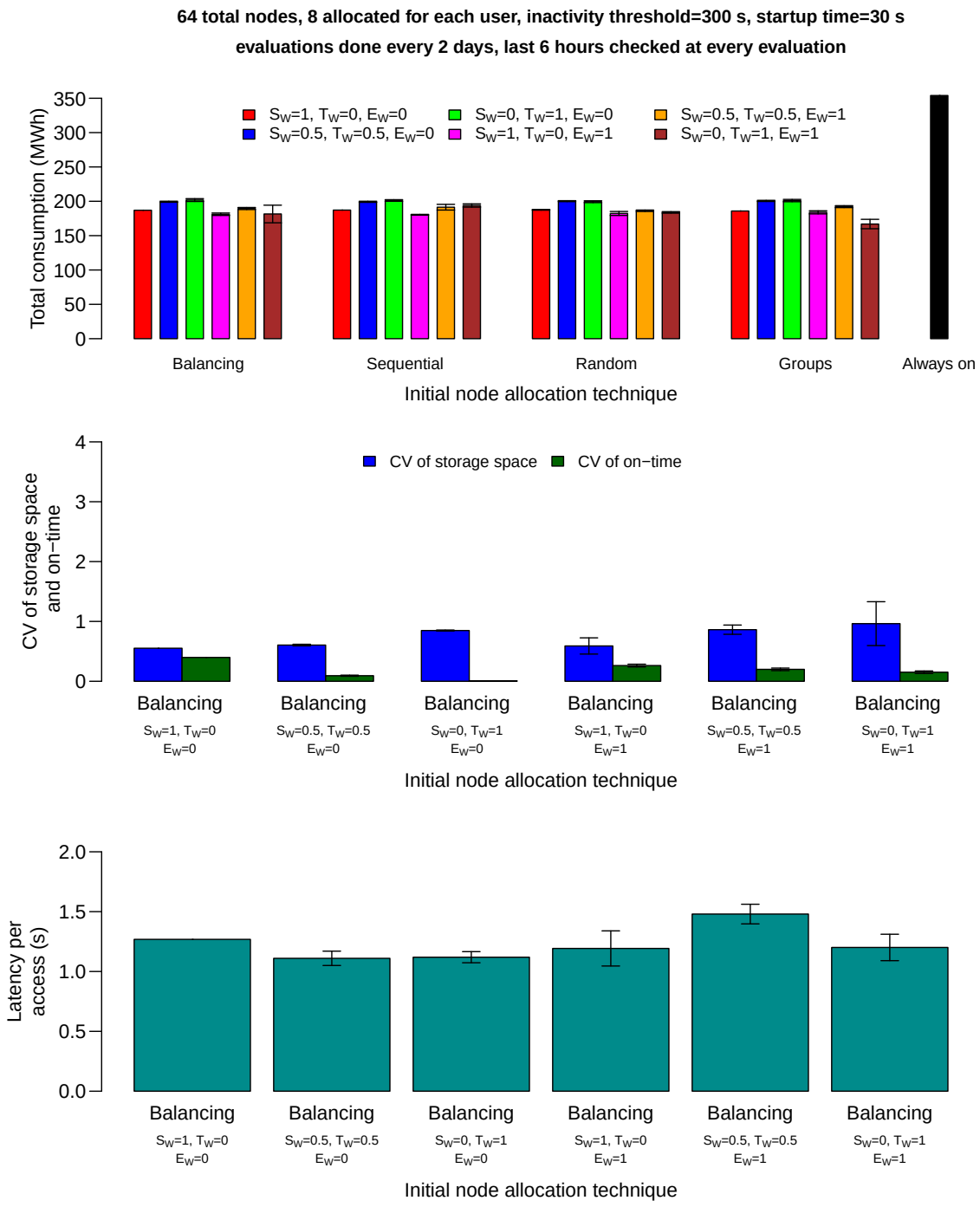


FIGURE 3.5.5: Initial technique vs energy consumption, load balance and latency per access (Dynamic Greedy scheme - Hornet cluster)

less energy compared to the cases where the energy consumption weight is set to zero. Another observation we can make is that the cases where the on-time balancing weight (T_W) is non-zero and energy consumption weight (E_W) is set to zero consume more energy. In the Hornet cluster workload, storage system access lengths vary significantly. As a result, while trying to balance the on-time across the storage nodes, it is more likely to transfer users from one node to another. Storage space usage among the users vary as well, though not as much as the access lengths.

In the second sub-plot of Figure 3.5.5, we can see how the CVs of storage space and on-time change when the storage space, on-time and energy consumption weights are varied. Regardless of which allocation technique is chosen initially (balancing, sequential, random or grouping) the CVs of the storage space and on-time are similar. Therefore, we only show the results for cases where balancing technique is the initially chosen technique. We can see that the CV of the storage space is the smallest when the storage space balancing weight (S_W) is set to one, regardless of the energy consumption weight (E_W). As the on-time balancing weight (T_W) is increased, the CV of the storage space increases as well. The same holds true for the CV of the on-time. When the on-time balancing weight (T_W) is set to one, the CV of the on-time is the smallest regardless of the energy consumption weight (E_W). As the storage space balancing weight (S_W) is increased, the CV of the on-time increases as well.

We can see how latency per storage access changes with varying storage space, on-time and energy consumption weights in the third sub-plot of Figure 3.5.5. Again, as it does not matter which allocation technique is initially chosen, we only show the results for cases where the balancing technique is the initially chosen technique. The latency measurements are almost the inverse of the energy consumption measurements in the first sub-plot of Figure 3.5.5. In the first sub-plot of Figure 3.5.5, we have seen that

when the energy consumption weight (E_W) is set to one, the energy consumption goes down. This means that a storage access will most likely be made on a turned-off node, increasing latency per access. We have also seen in the first sub-plot of Figure 3.5.5 that when on-time balancing weight is non-zero and the energy consumption weight (E_W) is set to zero, the energy savings are less. This will in turn decrease the chance of an access to be made to a turned-off node, decreasing latency per access.

We now look at the effect of varying the evaluation frequency on the total energy consumption, load balance and latency per access. The evaluation frequency is varied between 2, 10 and 20 days and the control period is fixed at 6 hours. Since the initial allocation technique does not change the results, we initially start with the balancing technique with both storage space and on-time balancing weights (S_W and T_W) set to 0.5 and energy consumption weight (E_W) set to zero. As shown in Figure 3.5.6, total energy consumption decreases, and consequently latency per access increases, by a very small margin as the evaluation frequency becomes longer. This is specific to Hornet cluster data as access lengths are long and the system is underutilized in the early stages. This has a very small effect on the energy consumption and latency per access. We also see that the CVs of the storage space and on-time is almost not affected by the evaluation frequency.

Figure 3.5.7 shows the effect of varying the control period (between 6, 12 and 24 hours) on the total energy consumption, load balance and latency per access. In this test case the evaluation frequency is fixed at 2 days. The results in Figure 3.5.7 indicate that varying the control period does not affect the total energy consumption, load balance or latency per access.

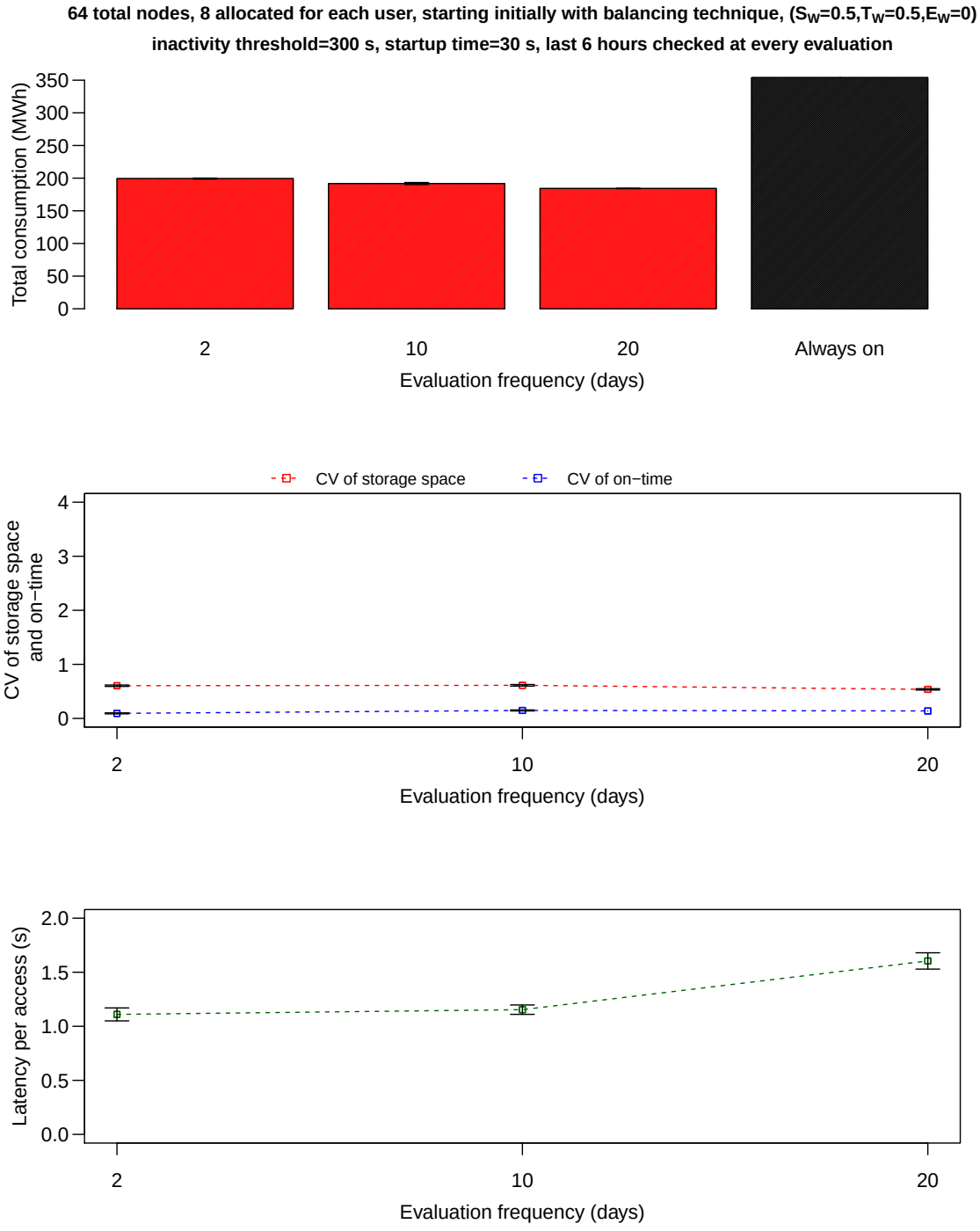


FIGURE 3.5.6: The effect of varying the evaluation frequency (Dynamic Greedy scheme - Hornet cluster)

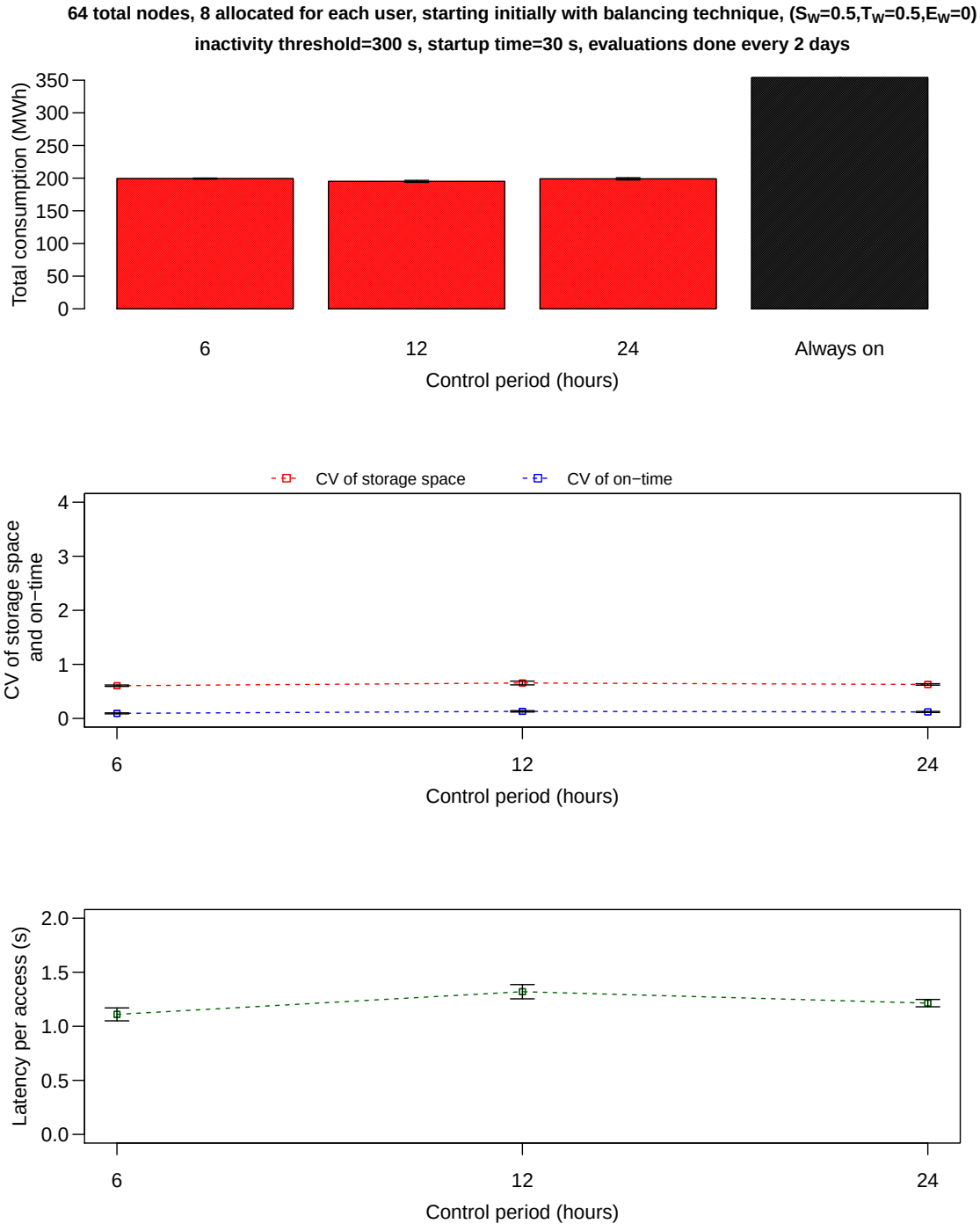


FIGURE 3.5.7: The effect of varying the control period (Dynamic Greedy scheme - Hornet cluster)

3.5.5 Correlation-based Scheme

In this section, we evaluate the Correlation-based scheme and find out how effective it is in terms of energy consumption, load balance and latency per access. Correlation-based scheme starts with one of the four node allocation techniques (balancing, sequential, random or grouping) similar to the Dynamic Greedy scheme. Unlike the Dynamic Greedy scheme, it checks the correlations between users at evaluation points and allocates the same storage nodes for correlated users instead of changing the initial node allocation technique. As a matter of fact, initial node allocation technique is used if only a user accesses the storage system for the first time. During the following tests, we also try to understand if the initial technique chosen has any effect on the energy consumption, load balance or latency per access.

Correlation-based scheme also has *evaluation point* and *control period* parameters. We have already shown the effect of varying these two parameters for the Dynamic Greedy scheme; therefore, we are not repeating the same tests for Correlation-based scheme. *Similarity threshold* is a parameter that is unique to the Correlation-based scheme. We look at the effect of this parameter on the energy consumption, load balance and latency per access as well.

To start with, we test the Correlation-based scheme with different initial techniques, where the total number of storage nodes in the system is 64 and the number of nodes allocated for each user is 8, as shown in Figure 3.5.8. The evaluations are done every 2 days and the storage accesses in the last 6 hours are checked. Each measurement is the average of five runs. Since energy consumption weight is not important in Correlation-based scheme, it is fixed at zero in this test case. Due to space considerations, we show only the GRID5000 workload evaluations of the

Correlation-based scheme, as other workloads have similar results.

Figure 3.5.8 shows that Correlation-based Scheme saves more than half of the energy consumed in the stock case regardless of the initial node allocation technique. The percentage of energy savings are as high as 60%. We observe that the initial technique chosen does not have a significant effect on the energy consumption.

The CVs of the storage space and on-time are shown in the second sub-plot of Figure 3.5.8. As the initial technique chosen for the Correlation-based scheme does not really matter, we show results for the balancing technique with varying storage-space and on-time balancing weights. As the storage space balancing weight (S_W) is increased, the CV of the storage space decreases. The same is true for on-time.

The third sub-plot in Figure 3.5.8 shows how latency per access is affected when the initial technique is balancing with varying storage-space and on-time balancing weights. In GRID5000 workload, the storage space usage of users are close to each other. Therefore balancing technique with storage-space balancing weight set to one will not effect the node allocations as much as other techniques do. We observed that GRID5000 workload has accesses of varying lengths, particularly until the first evaluation point. As a result, if both the storage space balancing and on-time weight are set to 0.5, then the algorithm will not try to break already balanced storage-space distribution, while at the same time trying to balance on-time. This will in turn result in more transfers and consequently more latency. This feature of the GRID5000 workload is similar to that of Google trace in Figure 3.5.2, except that Google trace had varying storage space usage among the users where access lengths did not vary that much.

Figure 3.5.9 shows the effect of varying the similarity threshold on the total energy consumption, load balancing and latency per access. Evaluations are done every 2

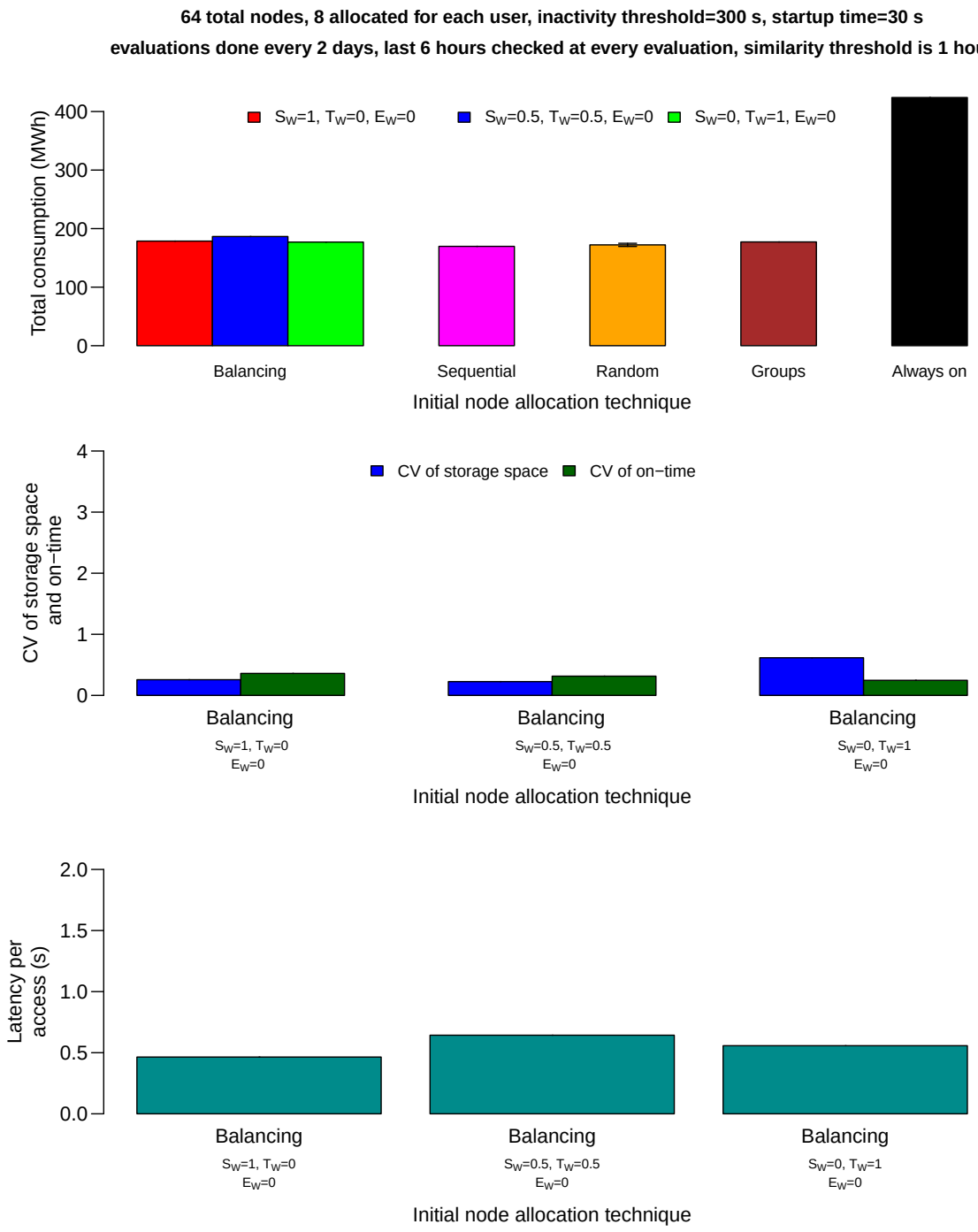


FIGURE 3.5.8: Initial technique vs energy consumption, load balance and latency (Correlation-based scheme - GRID5000 workload)

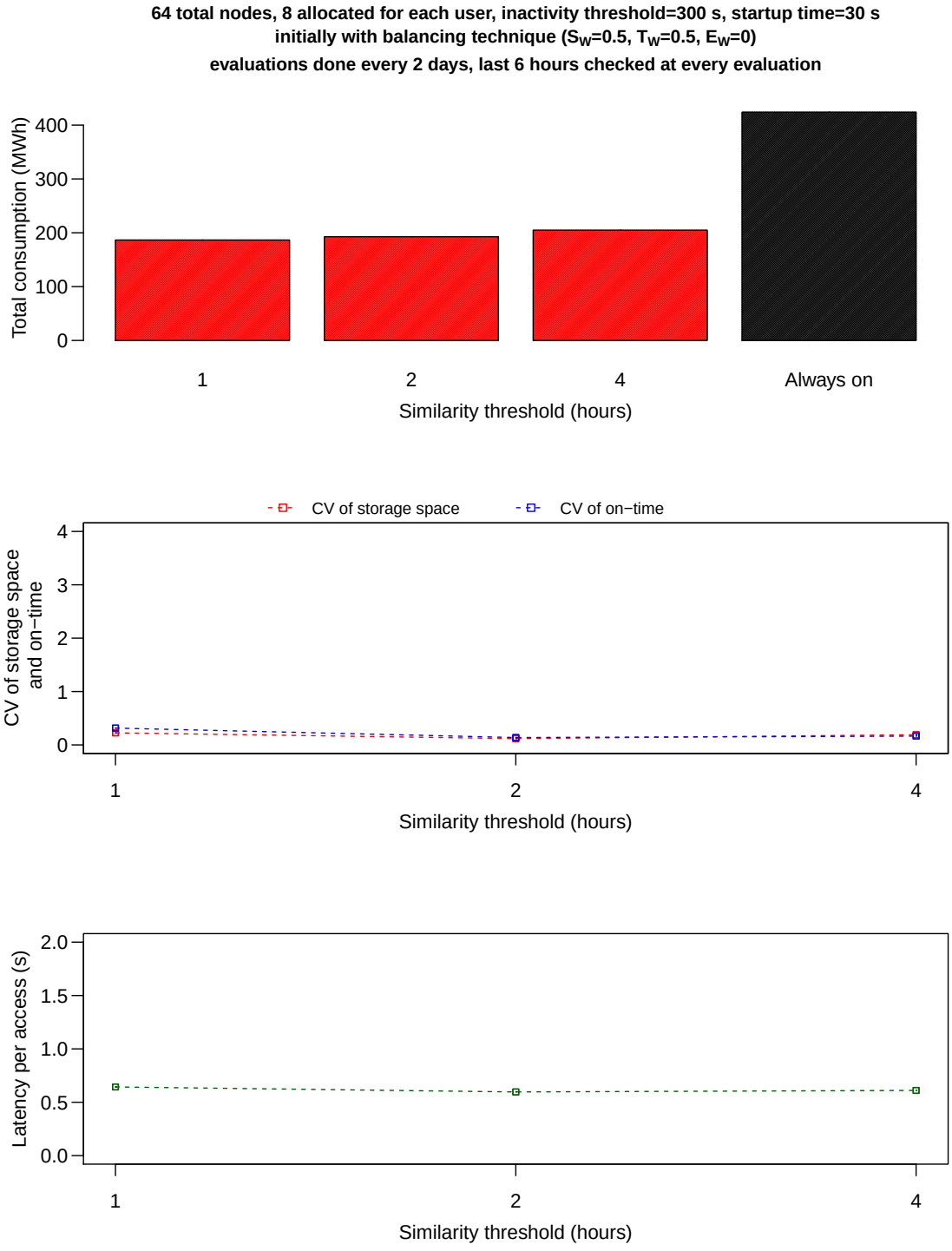


FIGURE 3.5.9: The effect of varying the similarity threshold (Correlation-based scheme - GRID5000 workload)

days and the last 6 hours are checked at every evaluation. Balancing technique with both storage space and on-time balancing weights set to 0.5 is used initially. Similar to the inactivity threshold, varying the similarity threshold has almost no effect on the standard deviations of the storage space and on-time or latency per access. When the similarity threshold is increased, total energy consumption increases slightly. The higher the similarity threshold is, the less likely it is to find correlated users. As a result, energy consumption goes up with higher similarity threshold.

3.5.6 Validating Mathematical Model

We have validated the mathematical model presented in Section 3.3 against the experimental results of our approach using data from Hornet HPC system. We can estimate the energy consumption, load balancing and latency per access values for each workload by using the mathematical model presented in Section 3.3. We validate the mathematical model using the subset of test parameters as shown in Table 3.5.2. For brevity, we show only the Hornet data, though the other workloads show similar results. For the Hornet data that we show here for this validation, we have primarily used a Generalized Extreme Value interjob arrival model - i.e. $F(t) = e^{-g(t)}, \quad where \quad g(t) \quad = \quad [1 + (\frac{t-\mu}{\sigma}) * \xi]^{\frac{-1}{\xi}}$ and $\xi, \quad \mu$ and σ are distribution parameters.

Tables 3.5.3, 3.5.4 and 3.5.5 show that the mathematical model we presented is able to estimate the test result parameters (CV of storage space, CV of on-time , energy cost and latency per access) accurately in most cases. The storage space usage of a user per storage node does not change over time; therefore, the chance of estimating it accurately is high. Our model estimates on-time value and energy

Test Parameter	Values
Total number of nodes	64
Number of nodes allocated for each user	8
Allocation methods	Fixed, Dynamic Greedy, Correlation-based
Low-energy mode	Turn Off
Inactivity threshold	300 s
Startup time	30 s
Similarity threshold	3600 s
Storage space and on-time balancing weights	0.5, 0.5, 0
Power consumption per node	300 W
Workloads	Hornet

TABLE 3.5.2: Mathematical model validation parameters

consumption of each storage node accurately as well, as estimating them involves all of the parameters captured by our model except storage space usage. Latency per access is highly dependent on the arrival rate of *interarrivals* and our model does the best attempt to capture these interarrivals in order to estimate the latency per access.

	Balancing				Sequential			
	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)
Simulation	0.539	0.312	195.65	1.0348	0.757	0.382	181.01	1.1833
Model	0.543	0.3145	195.62	1.1657	0.7628	0.3849	180.93	1.1669
Error	0.74%	0.80%	0.02%	12.65%	0.77%	0.76%	0.04%	1.39%

	Random				Groups			
	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)
Simulation	0.9	0.394	179.81	1.5673	1.129	0.351	183.78	1.0367
Model	0.9069	0.3973	179.753	1.2496	1.1381	0.3539	183.711	1.0539
Error	0.77%	0.84%	0.03%	20.27%	0.81%	0.83%	0.04%	1.66%

TABLE 3.5.3: Mathematical model validation results for Fixed scheme

	Starts with Balancing				Starts with Sequential			
	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)
Simulation	0.605	0.0932	199.36	1.11008	0.6186	0.0956	199.273	1.12687
Model	0.6031	0.1006	199.913	1.0424	0.6142	0.1	200.254	0.9944
Error	0.31%	7.94%	0.28%	6.1%	0.71%	4.6%	0.49%	11.76%

	Starts with Random				Starts with Groups			
	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)
Simulation	0.614	0.0926	200.199	1.14529	0.63	0.086	200.558	1.09082
Model	0.6092	0.0923	199.577	1.022	0.6382	0.08	201.176	1.0311
Error	0.78%	0.32%	0.31%	10.76%	1.3%	6.98%	0.31%	5.47%

TABLE 3.5.4: Mathematical model validation results for Dynamic Greedy scheme

	Starts with Balancing				Starts with Sequential			
	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)
Simulation	2.059	0.35	120.842	1.27998	2.108	0.671	105.857	1.27017
Model	2.0755	0.3523	120.798	1.5492	2.1244	0.676	105.826	1.2532
Error	0.80%	0.66%	0.04%	21.03%	0.78%	0.75%	0.03%	1.34%

	Starts with Random				Starts with Groups			
	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)	Storage CV	On-time CV	Energy cost (MWh)	Latency per access (s)
Simulation	2.056	0.638	103.395	1.36857	2.052	0.835	107.887	1.30135
Model	2.072	0.6427	103.381	1.5413	2.0678	0.842	107.851	1.249
Error	0.78%	0.74%	0.01%	12.62%	0.77%	0.84%	0.03%	4.02%

TABLE 3.5.5: Mathematical model validation results for Correlation-based Scheme

3.6 Summary

In this chapter, we presented three different energy-aware node allocation methods that take advantage of storage network incast and exploit user metadata for allocating cloud storage system resources for each user, while adhering to load-balancing parameters on demand. We also presented a mathematical model to estimate the outcome of the proposed methods and showed that the estimations of this model closely match with the simulation results. Each method has been evaluated both theoretically and through simulation-based tests using real-world workloads. Theoretical analysis results show that our proposed methods can provide 2-approximation solutions for minimizing energy consumption and balancing system load (storage space usage or on-time). Simulation-based tests show that the energy savings are as high as 60%. The simulation-based tests further show that as the storage system moves from the Fixed scheme to Dynamic Greedy scheme and then to Correlation-based scheme, energy savings, latency per access and coefficient of variation of the storage space and on-time increase. Therefore, these schemes can be implemented in a cloud storage system depending on parameters of importance (energy consumption, latency or load balancing). As an example, in a storage system where energy consumption is the primary concern, Correlation-based scheme can be used. Similarly, in a storage system where load balancing and energy consumption are equally important, Dynamic Greedy scheme can be used. We should also point out that the latency per access values for any of the methods we proposed were usually less than a second, which should be acceptable for most of the cloud storage applications.

Our methods are different from related studies as we classify and place users; rather than classifying and placing data. Additionally, the methods we propose take

load-balancing and data transfer costs into account, which is not included in many related studies. We also have a lightweight algorithm to predict future which is another concept that is missing in related studies, as they mostly try to predict future reactively by monitoring the system with complex mechanisms and possibly introducing overhead as a result of this.

Chapter 4

Metadata Performance

In this chapter, we describe our optimization techniques in order to tackle the metadata performance problem in large-scale storage systems. We propose optimization techniques to improve the performance of two common metadata-intensive operations; creating big number of files and reading a directory with big number of entries. As mentioned previously, we apply our optimization methods to an existing parallel file system integrated with object-based storage, Ohio Supercomputing Center's PVFS-OSD implementation.

The rest of this chapter is organized as follows: We present related research studies in Section 4.1. Section 4.2 shows how directories can be mapped to object storage collections and Section 4.3 explains lazy creation of objects. Then, we present experimental evaluation results in Section 4.4 and we concluded this chapter in Section 4.5.

4.1 Related Work

There have been several studies in terms of improving the performance of data operations in distributed systems. OSD+ [34] presents an object storage model that adds dedicated directory objects to T10 OSD standard [124]. Parallel file systems such as PanFS [131] and Lustre [40] improve the performance of data and metadata operations by using object storage features. Ceph [129] scales metadata by decoupling it from data and by distributing objects across storage devices.

There also have been studies trying to improve the scalability of directory operations in storage systems. GIGA+ presents a distributed directory service based on PVFS that stripes large directories over a number of servers in the system while at the same time balancing the load on all servers [100]. Yang et al. supports extensible hashing and the splitting strategy of GIGA+ in OrangeFS [21], new version of PVFS, to scale the directory metadata operations [137].

4.2 Collections as Directories

In this section, we propose our first optimization technique; ***representing traditional directories as collection objects in OSDs***. We describe two different configurations of the existing PVFS-OSD implementation and explain how we apply our optimization techniques to them.

4.2.1 Configuration 1: OSDs as I/O Servers

In this configuration, OSD servers are only responsible for I/O operations and for storing objects; while metadata and directory operations are managed by the PVFS

metadata servers.

Current metadata operation methods in Configuration 1

In this section, we start with a discussion of how various metadata operations - create file or directory, insert file and stat directory - are currently implemented in Configuration 1.

***Create a file and insert it into a directory** In order to create a new file, PVFS client sends a *create* command to both OSD I/O server and PVFS metadata server. OSD I/O server creates data object(s) and PVFS metadata server creates a metadata object for this file and they return data and metadata identifiers to the client. To create a directory, the client sends a *create directory* command to PVFS metadata server only. To insert a file into a directory, PVFS metadata server performs a lookup on the directory path and if the directory object exists, its identifier is returned to the client, which in turn inserts data and metadata identifiers into the directory object. Creating a file "sample" and inserting it into a directory "foo" is illustrated in Figure 4.2.1.

***Stat the entries of a directory** If a client wants to stat the entries of a directory, at first PVFS metadata server performs a lookup on the directory path as shown in Figure 4.2.2. As discussed in Section 4.2.1, each directory object stores data and metadata object identifiers corresponding to each directory entry. Thus, the client reads data and metadata identifiers stored in that directory object. For each metadata identifier, it contacts PVFS metadata server to retrieve metadata and

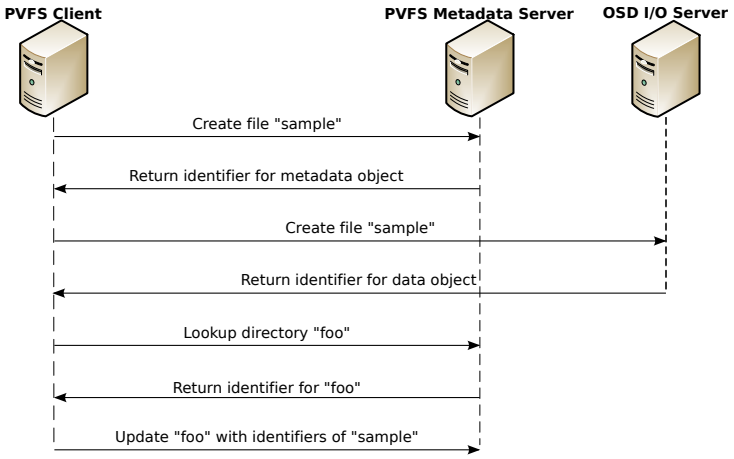


FIGURE 4.2.1: Current method to create a file and insert it into a directory in Configuration 1

for each data identifier, it contacts OSD I/O server(s) to retrieve logical size that is stored in a specific attribute. This process is illustrated in Figure 4.2.2 for directory "foo".

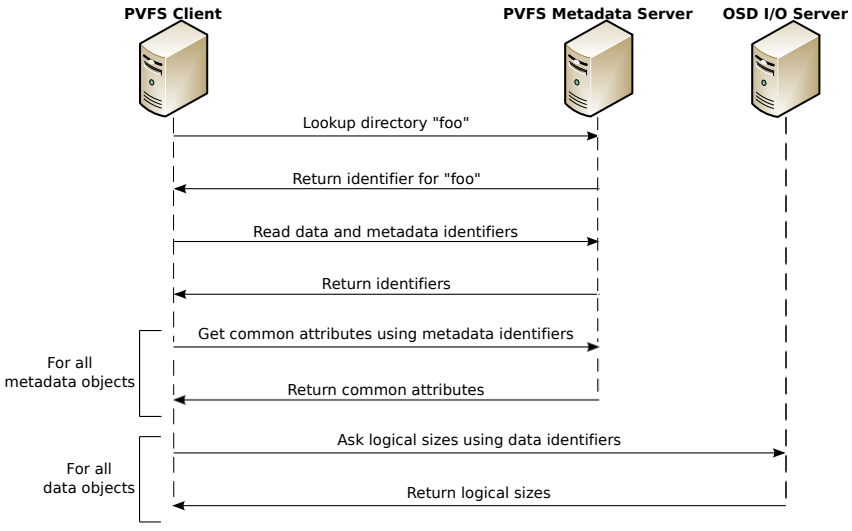


FIGURE 4.2.2: Current method to stat a directory in Configuration 1

Optimizing metadata operations by representing directories with collections

As can be inferred from Figure 4.2.2, communicating with the OSD I/O server each time to retrieve logical sizes is very inefficient. In order to alleviate this problem, we are proposing using OSD collections to represent directories on OSD I/O servers in addition to the directory objects on PVFS metadata servers and using the *get_member_attributes* primitive in the OSD standard [124] to retrieve logical sizes of all data objects in a collection at once.

OSD standard [124] includes four different types of data structures; root object (descriptive data about entire OSD), partitions (encloses collections), collections (logical grouping of user objects) and user objects (any data or metadata object). *Get_member_attributes* primitive finds data objects in a collection and retrieves the desired attributes (i.e. logical size) from all objects in a single communication step with the client.

Creating a file with our optimization is the same with the file creation procedure shown in Figure 4.2.1. Inserting a file into a directory differs from what is shown Figure 4.2.1 though. Metadata and data object identifiers are inserted into the OSD collection object, in addition to the directory object on the PVFS metadata server.

Creating a directory "foo", inserting file "sample" into it and performing a stat on "foo" is shown in Figure 4.2.3.

4.2.2 Configuration 2: OSDs as I/O and Metadata Servers

In this configuration, OSDs are responsible for all I/O, metadata and directory operations in the system; whereas PVFS is only used as the client.

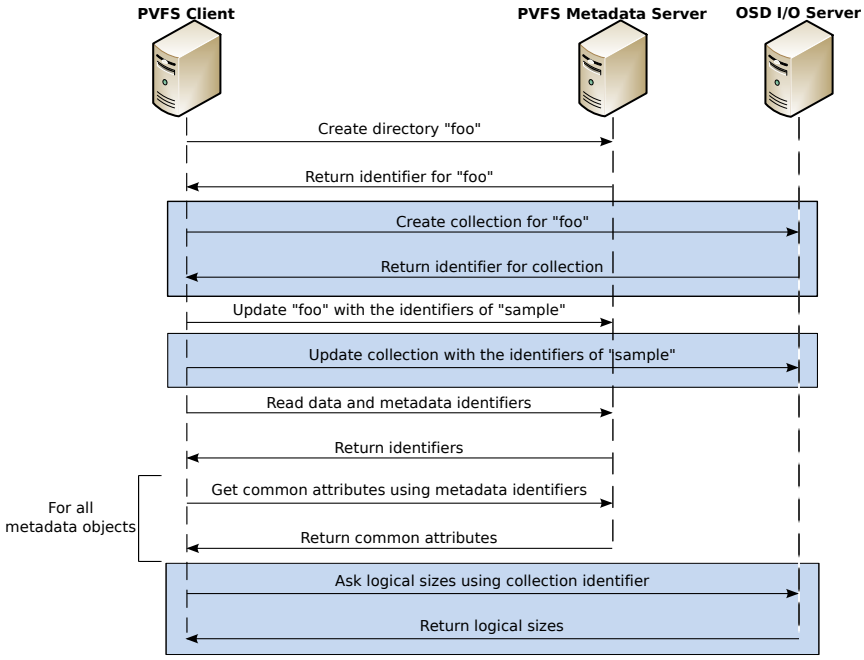


FIGURE 4.2.3: Proposed method to create a directory, insert a file into it and stat it in Configuration 1

Current metadata operation methods in Configuration 2

In this section, we start with a discussion of how various metadata operations - create file or directory, insert file and stat directory - are currently implemented in Configuration 2.

***Create a file and insert it into a directory** To create a new file, PVFS client sends a *create* command to the OSD I/O server, which in turn creates data object(s) and returns identifier(s) to the client. In Configuration 2, metadata is stored on OSDs using two different methods; using a dedicated OSD metadata server or storing it in data objects [29]. We chose the latter since it is more flexible and efficient. To create a directory, the client sends a *create directory* command to OSD

metadata server. The name and identifier of each entry are stored in directory object attributes using two different methods [28]. In the first method, directory object is locked at first, entry name and identifier are inserted into corresponding attribute and the lock is released. In the second method, atomic compare-and-swap primitive is used to insert name and identifier into an attribute [53]. We chose the former since it was more readily available and using the first or second method to insert a directory entry into a directory object would not create a difference in terms of the performance of the optimization we are proposing. Creating a file "sample" and inserting it into directory "foo" is illustrated in Figure 4.2.4.

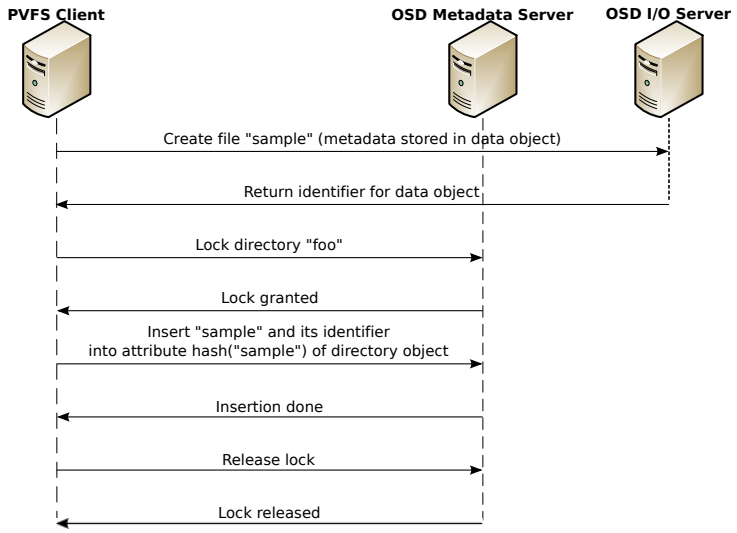


FIGURE 4.2.4: Current method to create a file and insert it into a directory in Configuration 2

Previous work [28] shows that when OSDs serve as both I/O and metadata servers, they can store the directory entries in the directory object attributes. Each entry's name is hashed to map that entry to an attribute that stores entry's name and identifier as illustrated in Figure 4.2.5 for "file x", "file y" and directory object with identifier n.

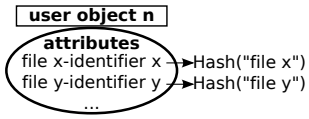


FIGURE 4.2.5: Current method to store the entries of a directory in a directory object in Configuration 2

***Stat the entries of a directory** To stat the entries of a directory, OSD metadata server performs a lookup on the directory path at first returning the identifier to the client. The client then reads the attributes of the directory object. As discussed in Section 4.2.2, each attribute stores directory entry names and identifiers. Using these, the client retrieves data object attributes from the OSD I/O server. This process is illustrated in Figure 4.2.6 for directory "foo".

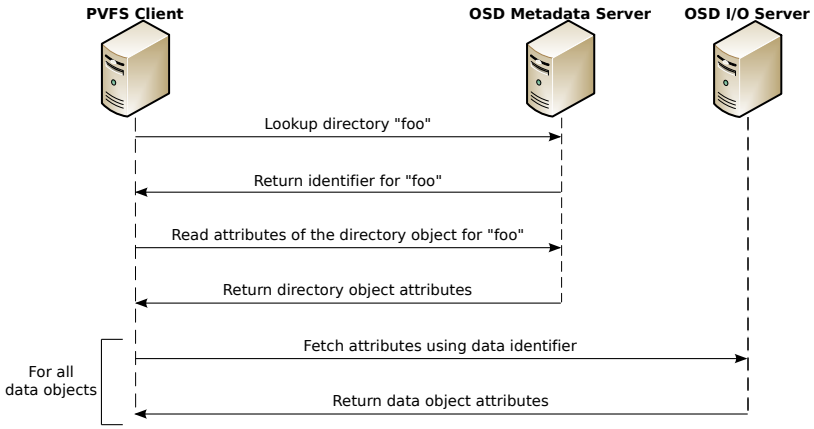


FIGURE 4.2.6: Current method to stat a directory in Configuration 2

Optimizing metadata operations by representing directories with collections

As can be inferred from Figure 4.2.6, multiple communication steps between OSD I/O server and the client is very inefficient. To alleviate this problem, we are proposing replacing directory objects on OSD Metadata servers shown in Figure 4.2.5 with

collections on OSD I/O servers and using *get_member_attributes* primitive to retrieve the attributes of all objects in a collection in a single communication step. Since directory objects on OSD Metadata servers are replaced with collections on OSD I/O servers, OSD metadata servers are not necessary anymore.

It is important to note that, in our proposed configuration OSD I/O servers are responsible for storing both the directory objects and the data objects. Thus there is no need to have a dedicated OSD Metadata server in our proposed configuration.

Creating a file with our optimization is the same with the file creation procedure shown in Figure 4.2.4. While inserting a file into a directory, data object identifier is inserted into the OSD collection object.

To stat the entries of a directory, PVFS client calls *get_member_attributes* on that directory's collection object. As a result, data object attributes can be retrieved from the OSD I/O server at once. Creating a directory "foo", inserting file "sample" into it and performing a stat on "foo" is shown in Figure 4.2.7.

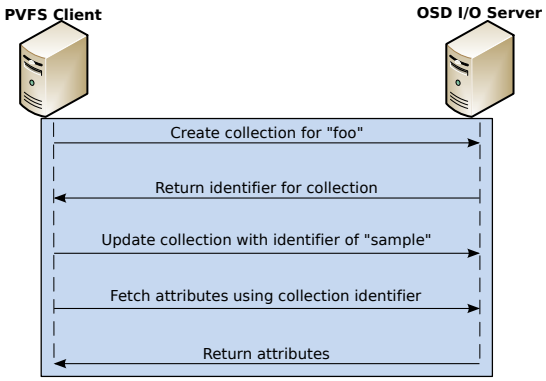


FIGURE 4.2.7: Proposed method to create a directory, insert a file into it and stat it in Configuration 2

4.3 Post-creating Objects

The goal of our second optimization is to improve performance while creating large numbers of files at once. Creating a file in a parallel file system is an inherently slow process; because of the steps involved in it.

- Create a metadata object in the metadata server.
- Create data object(s) in I/O server(s).
- Update the metadata object with data handles.
- Create a directory entry.

As we can see from the steps above, $4n$ steps are required to create n files. If the input file is striped over k OSD I/O servers, then $3n + kn$ steps are needed. Previous studies [55, 42] tried to solve this problem by pre-creating data files in a parallel file system with a batch create operation. The idea is to *pre-create* a large number of data objects during system initialization and to use pre-allocated datafile handles while creating new metadata objects. In this case, the metadata server does not have to wait for newly created data file handles that are normally returned by I/O servers. If the number of pre-created objects falls below a certain threshold, a batch create function is invoked to have a sufficient amount of preallocated objects. Pre-creating decreases the network traffic and it is very useful in cases where many small files are created.

We propose ***post-creating*** that is similar to pre-creating and that works as outlined below.

- Create a metadata object in the metadata server.

- Create a directory entry.
- Create and write to data object(s).

Our approach benefits from the *create and write* primitive in the OSD standard. In the stock case, a data object is created immediately after the creation of the metadata object. The creation of the data object is delayed until it is accessed by a write operation while using the post-create optimization. The process of creating and inserting a file into a directory with and without post-create optimization is illustrated in Figure 4.3.1.

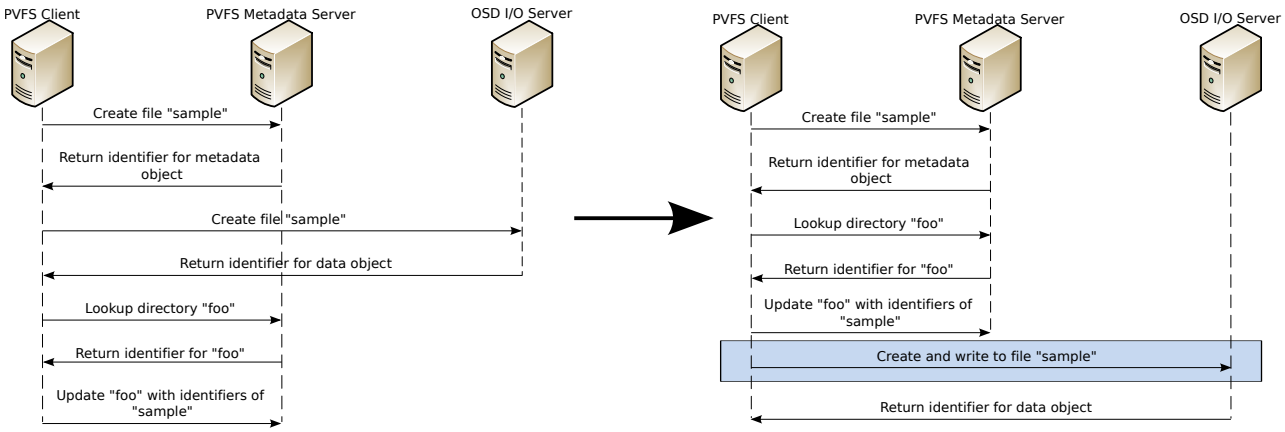


FIGURE 4.3.1: Current and proposed methods to create and insert a file into a directory in Configuration 1

Post-creating an OSD object greatly reduces the communication overhead between the PVFS client and the OSD I/O servers. Creating n data objects in the OSD I/O server requires $2n + kn$ communication steps compared to the $3n + kn$ steps in stock case, where k is the number of stripes for a given file.

One issue with post-creating an OSD object is that any object creation errors will not be identified until write time. Thus, an object creation error is interpreted as a write error rather than a file create error. We believe that this is actually the

correct interpretation, since object creation is a data operation rather than a metadata operation. As a result, object creation errors should reflect as write/data errors. Note that file creation (which is a metadata operation) is distinct from object creation.

4.4 Results

We conducted our evaluations on PVFS-OSD optimizations using an HPC cluster consisting of sixteen 64-bit x86 nodes with Linux Kernel 3.2 installed. Each node has 4 GB of RAM and 70 GB of storage, and the nodes are connected through a dedicated gigabit network switch. For experiments where the OSDs serve as I/O servers only, a separate machine is dedicated to serving as both PVFS metadata and directory server. We disabled the write caches of each node and flushed the read cache continuously during our tests; so that the I/O operations actually touch the disk. As mentioned before, OrangeFS 2.8.6 (new version of PVFS) and Ohio Supercomputing Center's OSD emulation is used in the tests.

4.4.1 Get_Member_Attributes

Configuration 1

In this section, we evaluate the speedup obtained from the *get_member_attributes* optimization where OSDs only serve as I/O servers in the system. Our current implementation for this test case works with a single OSD I/O server and a single PVFS metadata&directory server. We leave the implementation of *get_member_attributes* optimization for multiple OSD I/O servers where OSDs are only responsible for I/O

operations as part of our future work.

Figure 4.4.1 shows the average time per stat of a 1 MB file with or without *get_member_attributes* optimization. The results show that *get_member_attributes* can speed up stats up to 29 times. The stock case suffers from accessing OSD I/O server for each stat; whereas with *get_member_attributes*, all attributes are retrieved in a single call.

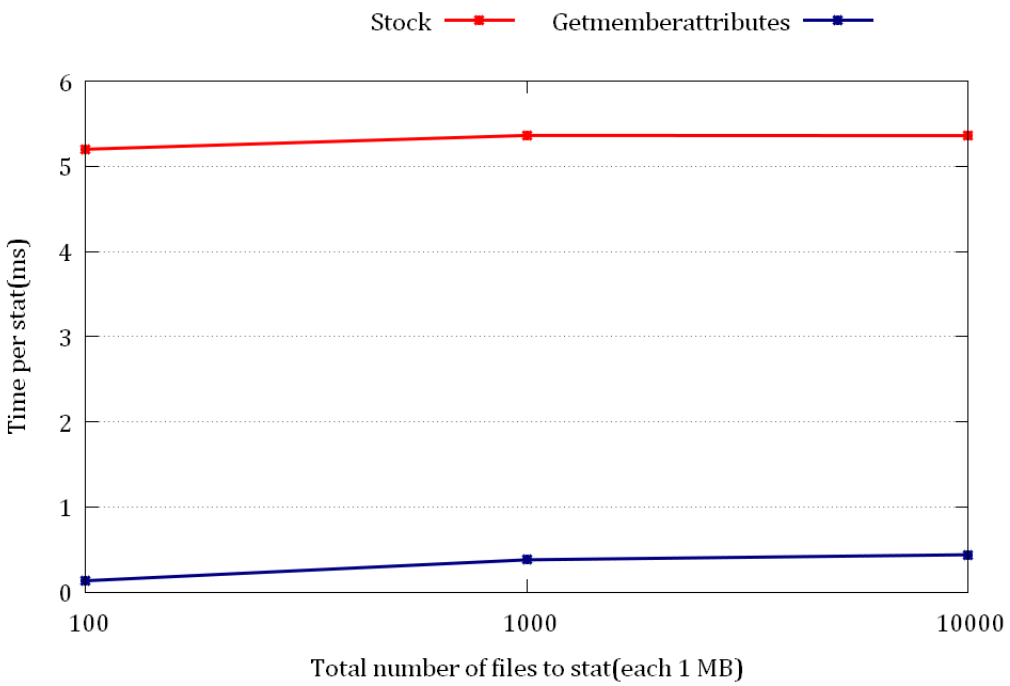


FIGURE 4.4.1: Time per stat for 100, 1000 and 10000 files (each 1 MB) with or without *get_member_attributes* optimization versus number of files to stat (single PVFS metadata & directory server and single PVFS client)

Configuration 2

In this section, we evaluate the *get_member_attributes* optimization for Configuration 2. A PVFS client creates 10000 files (each 1 KB) and then stats them with or without *get_member_attributes* optimization by varying the number of OSD I/O servers.

If *get_member_attributes* optimization is used in this case, then each directory is represented by a collection object in the OSD directory server and *get_member_attributes* primitive is called to stat a directory as shown in Figure 4.2.7. Otherwise, a separate call is initiated from the PVFS client to the OSD directory server to stat each file or directory as shown in Figure 4.2.6.

As can be seen in Figure 4.4.2, the average time to stat a file in a directory is improved by 60% and this improvement is nearly the same for different number of OSD I/O servers in the system. The speed-up for this case, where everything is in OSD, is not as big as the speed-up for the case where only I/O servers are OSDs as shown in Section 4.4.1; because in this case the overall performance of the system is already good before any optimization due to the fact that most of the communication is happening between OSDs. Still our optimization technique is able to improve overall system performance by 60%.

4.4.2 Post-create

Single client

In this section, we evaluate the efficiency of the *post-create* optimization In this test case there is one PVFS client, one PVFS metadata server and the number of the OSD I/O servers is varied. PVFS client writes ten thousand 1 KB files with and without *post-create* optimization and the average time per create is measured. The reason for not creating or reading more than ten thousand files in this test case is a limitation that is set by the Linux kernel block SCSI interface to the iSCSI protocol.

As can be seen in Figure 4.4.3, performance is improved by up to 10% for a single client. One can argue that 1 KB file size is too small to evaluate *post-create* optimiza-

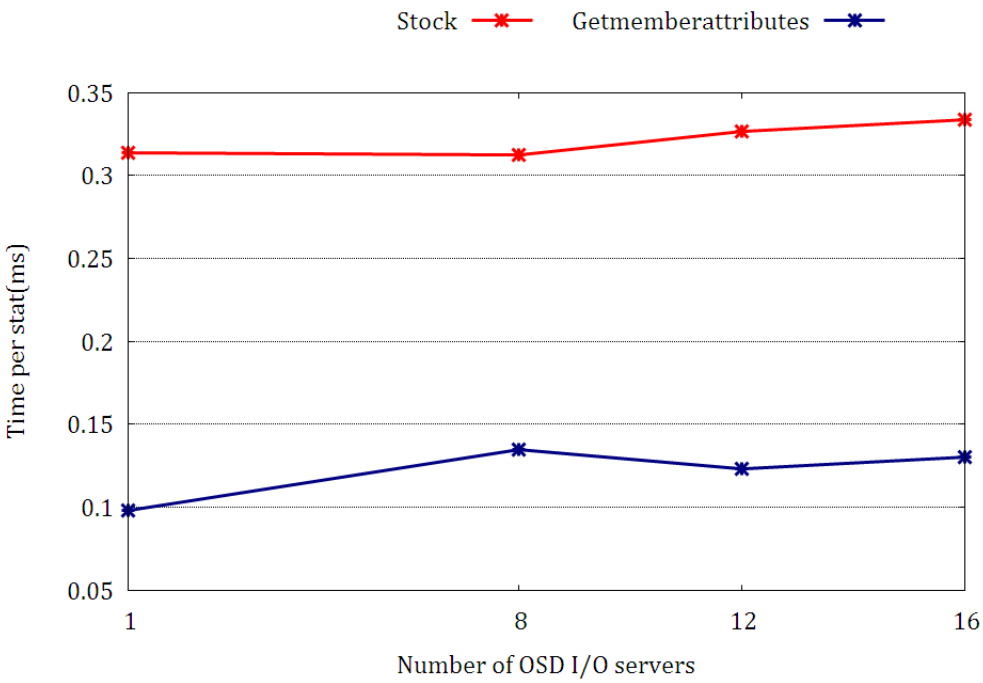


FIGURE 4.4.2: Time per stat for 10000 files (each 1 KB) with or without *get_member_attributes* optimization versus number of OSD I/O servers (single OSD directory server and single PVFS client)

tion. When this test is done with 1000 1 MB files on seven OSD I/O servers, *post-create* offers little benefit; since writes dominate total time. The *post-create* optimization works better with small files that can be commonly seen in HPC systems [59, 102]. As the total number of files to create increases when the number of OSD I/O servers is small, the OSD I/O servers will eventually become overloaded. This is the case where our *post-create* optimization pays off, which means that *post-create* optimization is suitable for I/O intensive workloads.

Multiple Clients

In this test case, multiple clients write hundred, one thousand and ten thousand 1 KB files to eight OSD I/O servers with or without *post-create* support and average

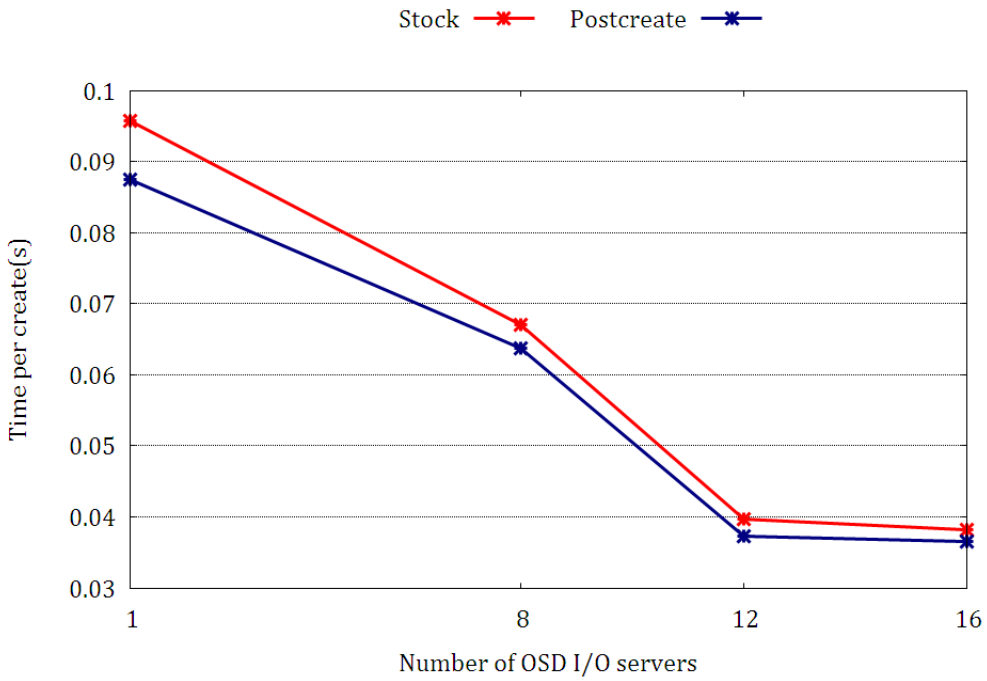


FIGURE 4.4.3: Time per create for 10000 files (each 1 KB) with or without *post-create* support versus number of OSD I/O servers (single PVFS metadata & directory server and single PVFS client)

time per create is measured.

Figure 4.4.4 shows that, as the number of clients concurrently creating objects increases, the performance gain from *post-create* optimization increases by up to 25%. This improvement in performance is comparable to the 19% improvement reported by using the *pre-create* optimization in prior work [42], that does not time the actual object creates that we include, since objects are batch created beforehand. In our approach, the objects are created at the same time as the write, so are included in our timing. We believe *post-create* can be further improved using stuffing (reducing the number of data objects allocated for small files) and coalescing (queuing operations in intensive workloads). Figure 4.4.4 also shows that *post-create* in a multiclient environment performs better with more OSD I/O servers, since it achieves

25% improvement with eight OSD I/O servers.

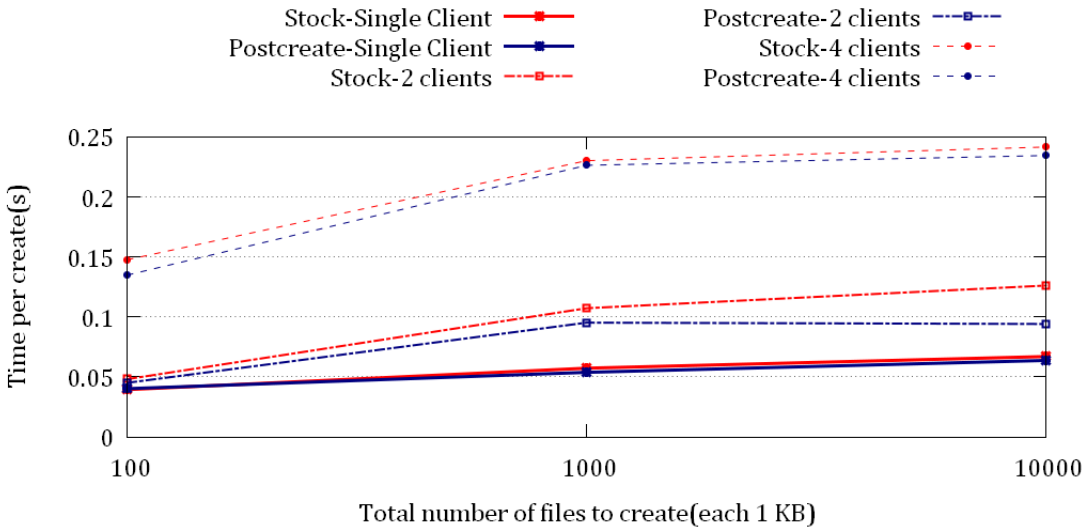


FIGURE 4.4.4: Time per create for 100, 1000 and 10000 files (each 1 KB) with or without *post-create* support for various number of clients (single PVFS metadata & directory server and eight OSD I/O servers)

4.5 Summary

In this chapter, we have presented two methods (using object collections as directories and post-creating objects) to improve the metadata performance of two common operations in distributed storage systems; creating a large number of files in a directory and reading from a directory with large number of files. Experimental evaluations show the efficiency of our approach.

Chapter 5

In-situ Computation on Object Storage

High-performance computing on large-scale data has become an important use case in recent years. There are various storage system solutions for end users to perform high-performance computation on large-scale data, while also providing data protection and concurrency between different users [12, 4, 19].

Clusters and cloud storage applications that perform computation on large-scale data typically do so on separate compute and storage clusters, since they have different architectural and functional requirements. In this architecture, however, large amounts of data needs to be transferred between storage and compute clusters to perform computation and to retrieve results back to the storage. Today, many storage systems store petabytes of data for various applications, such as climate modeling, astronomy, genomics analysis etc., and the amount of data stored in these systems is estimated to reach exabyte scale in the near future [63]. Therefore, moving big amounts of data between storage and compute nodes is not an efficient way of per-

forming computation on large-scale data anymore. Additionally, storing data both at the storage and compute sites increases storage overhead and with data replicated multiple times at both sites for resiliency, this overhead becomes even worse. Moving data between storage and compute nodes also increases the total energy consumption and network load.

On the other hand, there have been many efforts that have gone into improving storage interfaces and abstractions in order to store and access data more efficiently. As mentioned in Chapter 2, object-based storage is one of the most significant efforts in this area and many scaled-out storage systems today [40, 86, 20] are based on the object-based storage abstractions. We believe that, object-based storage features can be exploited to mitigate the data transfer overhead while performing computation on large-scale data. The computation framework in a cluster or cloud application can benefit from the intelligence of the underlying object-based storage to eliminate data movement while enabling in-place analytic capabilities. Consequently, the storage layer can be scaled while the computational framework remains lightweight. In this chapter of the thesis, we propose an example of this approach by integrating a computational framework, Hadoop [5], with the Ceph object-based storage system [129] and investigate the outcomes of our method. We also conduct performance evaluations using several benchmarks with various redundancy and replication policies and show that our implementation improves initial data copy performance of Hadoop by up to 96% and MapReduce performance by up to 20%. It is important to note that, both the computational framework and the object storage system can still be used as stand-alone systems in our approach, meaning that their normal functionalities are not impacted.

The rest of this chapter is organized as follows. We first present related studies

in Section 5.1. Then in Section 5.2, we present our approach to have data analytics capabilities on large-scale data. Section 5.2 gives evaluation results of the proposed method and Section 5.5 concludes this chapter.

5.1 Related Work

We have investigated related studies done on Hadoop in terms of improving the performance of data operations in distributed systems. Some studies analyzed and tried to improve Hadoop performance without integrating it with an underlying storage system. Shvachko et al. show the metadata scalability problem in Hadoop, by pointing out that a single namenode in HDFS is sufficient for read-intensive Hadoop workloads, while it will be saturated for write-intensive workloads [121]. Some related studies improved the performance of Hadoop by modifying its internal data management methods. *Scarlett* system proposes replicating data based on popularity, rather then creating replicas uniformly and causing machines containing popular data to become bottlenecks in MapReduce frameworks [31]. Porter analyzes the effects of decoupling storage and computation in Hadoop by using *SuperDataNodes*, servers that contain more disks than traditional Hadoop nodes, for the cases where the ratio of the computation to storage is not known in advance [106]. *CoHadoop* modifies Hadoop by co-locating and co-partitioning related data on the same set of nodes with the hints gathered from the applications [60]. *Maestro* identifies map task executions processing remote data as an important bottleneck in MapReduce frameworks and tries to overcome this problem by proposing a scheduling algorithm for map tasks that improves locality [72].

Hadoop is also integrated with cluster file systems in a number of studies, in order to analyze the outcomes of using cluster file systems for MapReduce applications. Tantisiriroj et al. integrate *PVFS* [45] with Hadoop and compare its performance to HDFS [125]. Ananthanarayanan et al. use *metablocks*, logical structures that support both large and small block interfaces, with *GPFS* to show that cluster file systems with metablocks can match the performance of Internet file systems for MapReduce applications [32]. *Lustre* can also be used as the back-end file system of Hadoop [24].

More recent work integrates OpenStack Swift with MapReduce framework for in-place data analytics [111]. This work, however, overrides OpenStack Swift's replication policy and performs worse in terms of performance due to the time reducers spend while renaming results. Cheng et al. present a storage tiering framework [47], CAST, that performs cloud storage allocation and data placement for data analytics workloads by leveraging the heterogeneity in cloud storage resources and within jobs in an analytics workload. Sevilla et al. present SupMR [117], where MapReduce input splits are created from data chunks rather than entire data, meaning that data is still copied to HDFS layer. Nakshatra [75] uses pre-fetching and scheduling techniques to improve the performance of data analytics jobs that are executed directly on archived data. However, data is still read and ingested into HDFS. Similarly, VNCache [99] and MixApart [90] use pre-fetching and scheduling techniques to ingest data to a cache on compute cluster. However, they still transfer data from the storage cluster to the compute cluster and they need to have mechanisms to maintain and clean the caches on compute nodes. There has been a study similar to ours to integrate Lustre with Hadoop [113], but hard links are only used for the intermediate output data of mappers to eliminate the HTTP overhead of the shuffle process. In our case, the input data of mappers is local and symbolic links are used for the input data to be ingested

into HDFS. Input data in HDFS is much larger in size compared to the output data of mappers. Therefore, our approach offers more performance improvement. Also in [113], a fast network interconnect is assumed to be readily available between the storage and compute nodes, which is not always the case. Our proposed approach is not dependent on the type of network interconnect. Yu et al. [138] present an implementation similar to our approach, where they try to enable in-situ data analytics on Lustre storage nodes. Our approach differs from this approach in that, we schedule map tasks to work on local data where this is not always the case in their proposed approach. Furthermore, we only learn the replica locations from the underlying storage system once when the Hadoop framework is initialized, whereas they query the Lustre metadata server for replica locations each time, which can be costly in terms of performance. And finally, they co-locate Lustre and Yarn by running Yarn in virtual machines and Lustre on the physical machine, where we run both on physical machines. VAS framework [136] is a similar study except that it does not always follow the replication policy of the underlying Lustre storage system and it co-locates data and computation using virtual machines. Wilson et al. presents RainFS to integrate MapReduce with HPC storage [135]. However, they consider network-attached storage only, while we have Hadoop daemons and underlying storage system located on the same physical node.

5.2 Integrating Object-Storage with a Computation Framework

In this section, we present our approach to integrate Hadoop with an object-based storage system - we use Ceph as our demonstration platform, but any object-based

storage system such as PVFS [45] or Lustre [40] could be used. As mentioned in Section 5.1, Hadoop consists of a computation layer that implements MapReduce framework and a storage layer, HDFS, that manages the underlying storage system. We modify Hadoop to perform in-place computation on large-scale data without moving or transferring data anywhere. The goal of our approach is to have a lightweight analytics (MapReduce) layer, while scaling the underlying storage system.

A typical Hadoop cluster is shown in Figure 5.2.1. In the storage layer, data and metadata are replicated and stored as *blocks* in the underlying datanodes. In the context of this discussion, a *block* is the smallest unit of data that can be stored in Hadoop. Namenode is responsible for metadata operations, such as keeping track of the block locations in the datanodes, collecting status reports from the datanodes and choosing datanodes to perform I/O operations on. Each datanode periodically sends heartbeats and block reports to the namenode to keep its global view updated. When a client wants to perform an I/O operation (read or write) in the system, it first communicates with the namenode to learn the locations of the data blocks for a read operation or to get a list of datanodes that will store the data blocks for a write operation. As soon as the client retrieves the required information from the namenode, it can communicate with the datanodes to perform the I/O operation. While reading, the client chooses one of the replicas (generally the closest one in terms of the network distance) to read the data. On the other hand, the write operation is performed in a pipeline, i.e. data is written to the first replica location at first, then it is forwarded from the first replica location to the second replica location and so on. The client finally receives an acknowledgement for a successful write operation. In the computation layer, a namenode is usually also a jobtracker, which assigns tasks to the tasktrackers (usually co-located with the datanodes) and keeps track of them.

When a MapReduce application is successfully completed, the jobtracker receives an acknowledgement from all the tasktrackers.

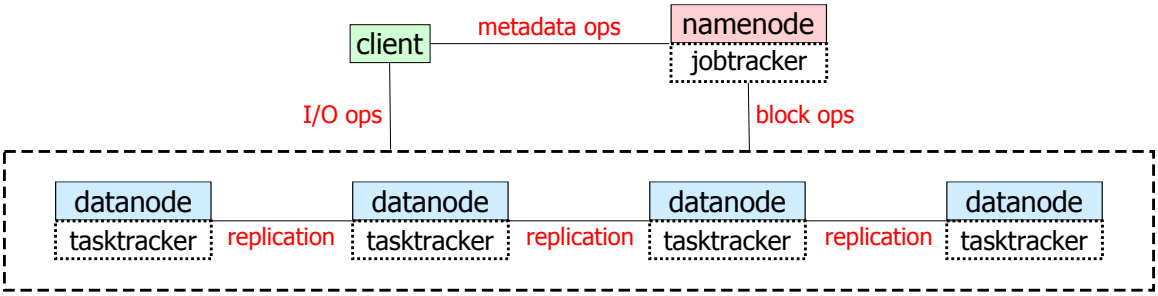


FIGURE 5.2.1: Hadoop architecture

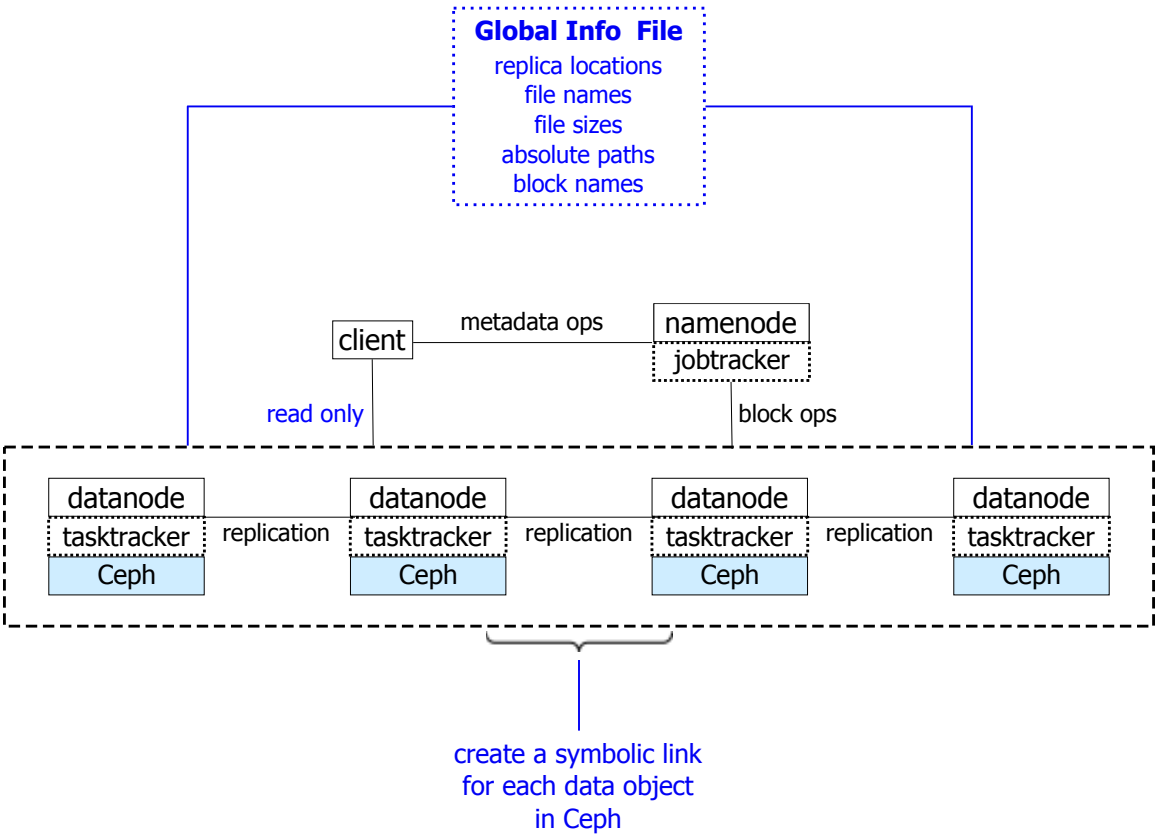


FIGURE 5.2.2: Proposed architecture

In our configuration, we have Hadoop datanode and tasktracker processes co-

located with the Ceph storage nodes as shown in Figure 5.2.2. Hadoop namenode and jobtracker processes can be also co-located with Hadoop datanode and tasktracker processes. Therefore, in our implementation the number of physical nodes used by Hadoop and Ceph are exactly the same and their daemons are co-located.

There can be multiple methods to integrate Hadoop with the underlying Ceph storage system. The most straight-forward approach would be to transfer object data from Ceph to HDFS. Another approach would be using Ceph as the back-end storage of Hadoop and implementing Hadoop data management policies there. Instead of these methods, we suggest an alternative approach, *creating symbolic links in HDFS for data that already exists in Ceph*. Creating symbolic links is a fairly fast operation and it alleviates the need to transfer data to the HDFS storage system. However, in order to create symbolic links for the existing Ceph data, we need to know the properties of these objects.

Our modified Hadoop implementation relies on Ceph to fetch information about already existing objects. In order to make Hadoop aware of the existing data in Ceph storage system, Ceph is scanned when Hadoop is initialized and the scanned information is stored in the *Global Information File*, which is distributed to all nodes in the system. Scanned information includes file names, replica locations, absolute paths, file sizes and pre-calculated block names.

- *File names* and *pre-calculated block names* are used by Hadoop to identify objects that already exist in Ceph. Pre-calculated block names are formed by hashing object name, object location and replication level in Ceph. If a block does not follow this naming convention, Hadoop will treat that block as a regular HDFS block and this will make it possible for Hadoop to preserve its normal

functionality. Meanwhile, we do not have to make all Ceph objects visible to Hadoop. Any object that is not scanned will not be in the *Global Information File* and therefore, will not be visible to MapReduce applications.

- *Replica locations* are of critical importance. As it is the case with any other storage system, Ceph has its own replica placement policy and we do not want to change this policy, as our ultimate goal is to perform in-place computation on existing data. Therefore, we follow the replica placement policy of Ceph storage system and perform computation on existing data without moving it anywhere else.

- *Absolute paths* are necessary while creating symbolic links to already existing Ceph data from Hadoop.

- Since we are not copying any data into HDFS, Hadoop does not know about the sizes of existing Ceph objects. We feed the *file sizes* from *Global Information File* to Hadoop, so that MapReduce operations work as expected.

The procedure to create and distribute the *Global Information File* is less than ten seconds for even 150 GB of data in Ceph and this is a one-time operation that is performed when Hadoop is initialized. Hadoop trusts the information in *Global Information File* and once Hadoop daemons are successfully started, Ceph daemons do not even have to run anymore. If new data is available in Ceph, Ceph can be scanned again to regenerate the *Global Information File*. Hadoop daemons will pick up the updated information after a restart. It is important to note that incrementally updating the *Global Information File* with each write operation in a write-heavy workload will be expensive as the existing data blocks will be scanned over and

over again. Our solution is geared towards updating the *Global Information File* asynchronously on already existing data in a write-once read-many workload, which is the optimal use case for MapReduce applications.

As a result, whenever a client wants to perform computation on already existing Ceph data, Hadoop namenode will learn the Ceph replica locations from the *Global Information File* and will create symbolic links on nodes where Ceph replicas are located. Absolute file paths, file names and pre-calculated blocks names will be used during symbolic link creation. When Hadoop datanodes read the symbolic links, file sizes will be used to read the correct amount of data. Another important design decision we had to make was to disable datanode block scanners. Hadoop datanodes are responsible for scanning data blocks they own and they report bad blocks to the namenode. Our symbolic links were picked up during block scans and to prevent this from happening, we disabled datanode block scanners. We describe the configuration changes for datanode block scanner in Section 5.3.1.

It is also important to note that we create symbolic links for data blocks only. Since we do not read any data into HDFS, Hadoop creates a metadata block of negligible size for a Ceph object. We also have most of the metadata we need for symbolic links in the *Global Information File*. Hadoop metadata creation is dominated by checksum calculation and reading no data from Ceph means having an empty checksum. As a result, Hadoop metadata block creation overhead is negligible. We leave checksum implementation as a future work item, but handling it without reading any data into HDFS is tricky as Hadoop and the underlying storage system might have different checksumming algorithms. As an example Hadoop uses CRC32, but if the underlying storage system uses another checksumming algorithm (e.g. MD5), this will at least require converting one checksum to another, which is very expensive, even if no data

is read into HDFS.

The last change we have made was to the scheduling policy of mappers. As we read no data from Ceph to HDFS and create symbolic links that point to existing data, mappers have to work with local data in order to function properly. We modified the MapReduce task scheduler, so that it assigns local map tasks if a MapReduce operation is being executed on symbolic links (existing Ceph data). This also requires that MapReduce split size to be the same with the split size of the underlying storage system. Configuration changes for the MapReduce split size are also described in Section 5.3.1. In this section, we first describe the experimental setup followed by the performance evaluation tests we have conducted. Then, we discuss the outcomes of the performance evaluation tests.

5.3 Experimental Setup

We conducted our experimental evaluations using five Google Compute Engine [12] instances. Each instance has two Intel Sandy Bridge vCPUs, 7.5 GB of memory and 250 GB of storage. We grouped the instances in an instance group in *us-central1-a* zone to simulate a cluster consisting of five nodes. Each instance had internal and external network access configured and we enabled passwordless SSH connection between the instances.

5.3.1 Hadoop Configuration Parameters

In order to have a fair comparison of our implementation against stock Hadoop, we made sure that the configuration parameters (dfs and mapreduce) of both are

exactly the same. Table 5.3.1 shows the configuration parameters we have used for the experimental evaluations.

Hadoop Configuration Parameter	**Values**
dfs.replication	2 or 3
dfs.datanode.max.receivers	81920000
dfs.datanode.socket.write.timeout	0
dfs.datanode.scan.period.hours	-1
mapred.child.java.opts	-Xmx6144m
mapred.task.timeout	0
mapreduce.map.output.compress	true
mapreduce.map.output.compress.codec	org.apache.hadoop.io.compress.GzipCodec
mapred.child.ulimit	unlimited
mapred.min.split.size	matches underlying storage (see below for explanation)

TABLE 5.3.1: Hadoop configuration parameters

Details on these configurations are available in the Hadoop documentation [14]. There are two important configuration parameters that we would like to explain further here - *dfs.datanode.scan.period.hours* and *mapred.min.split.size*. Since we are not really writing any data to HDFS, but, rather creating symbolic links to existing data, datanode scans catch these links. In order to make our approach work, we needed to disable datanode scans. Additionally, we needed to make sure that the minimum split size of MapReduce, *mapred.min.split.size*, matches that of the underlying storage system, so that they work on the same number of splits for a fair comparison.

5.3.2 Benchmarks

We have tested our proposed changes using Hadoop 1.1.2 stable version and Ceph 0.94 (Hammer) release. The benchmarks we have tested were *Grep, Wordcount,*

TestDFSIO and *TeraSort*. These benchmarks are commonly used to evaluate Hadoop frameworks and they have different characteristics in terms of the size of data they use or generate. *Grep* searches for a pattern in a potentially large file and generates a small set of output containing matches. *Wordcount* is similar, but it generates much larger output. *TestDFSIO* and *TeraSort* generate their own input data and perform their tests on the generated data. *TestDFSIO* performs basic I/O operations (read and write in our case) on the generated data. Number of files to perform I/O and size of the I/O operation are configurable parameters of *TestDFSIO*. *TeraSort* sorts data generated by the *TeraGen* benchmark and optionally, sorted data can be verified with *TeraValidate* benchmark. The size of data produced by *TeraGen* is also a configurable parameter.

5.4 Results

We have tested the proposed changes with the following parameters shown in Table 5.4.1.

Test Parameter	**Values**
Total number of nodes	3, 5
Replication levels	2 replicas, 3 replicas
Benchmarks	Grep, Wordcount, TestDFSIO, TeraSort
Input size per file	Grep (25 MB, 242 MB, 2.4 GB) Wordcount (29 MB, 286 MB, 2.8 GB) TestDFSIO (500 MB, 5000 MB, 15000 MB, 25000 MB, 50000 MB) TeraSort (1 GB, 10 GB, 50 GB)

TABLE 5.4.1: Test parameters

5.4.1 Grep

We first present the experimental evaluation results for the *Grep* benchmark. This benchmark searches for a pattern in a given file and extracts the occurrences of the given phrase to the resulting output file - so, it generates output much smaller in size compared to its input.

In our test scenario, we first generate 100 files that are equal to each other in size (25 MB, 242 MB or 2.4 GB). Stock Hadoop creates these files in HDFS and writes data to each. In our proposed implementation, these files already exist in Ceph and we just make Hadoop aware of these existing files through symbolic links during its initialization. Following the data creation, we run the *Grep* benchmark on the newly created data. We perform the test steps above for both 3 nodes with 2 replicas and 5 nodes with 3 replicas.

We first measure the total time it takes to copy data into HDFS at first and show the results in the upper sub-plot of Figure 5.4.1. As we can observe, our proposed implementation takes significantly less time than stock Hadoop to copy data into HDFS, because we are not really ingesting any data into HDFS. As the input size per file is increased from 25 MB to 2.4 GB, the time it takes to copy data into HDFS increases for stock Hadoop. As we are not writing any data to HDFS, that time stays constant in our implementation. We are able to achieve a 95% improvement in terms of initial data copy performance and as the input size becomes larger, this improvement will become even higher. We can also see that the number of replicas or total storage nodes in the system doesn't have a significant effect on the performance of the data copy phase.

We are not really targeting to improve the MapReduce performance as our op-

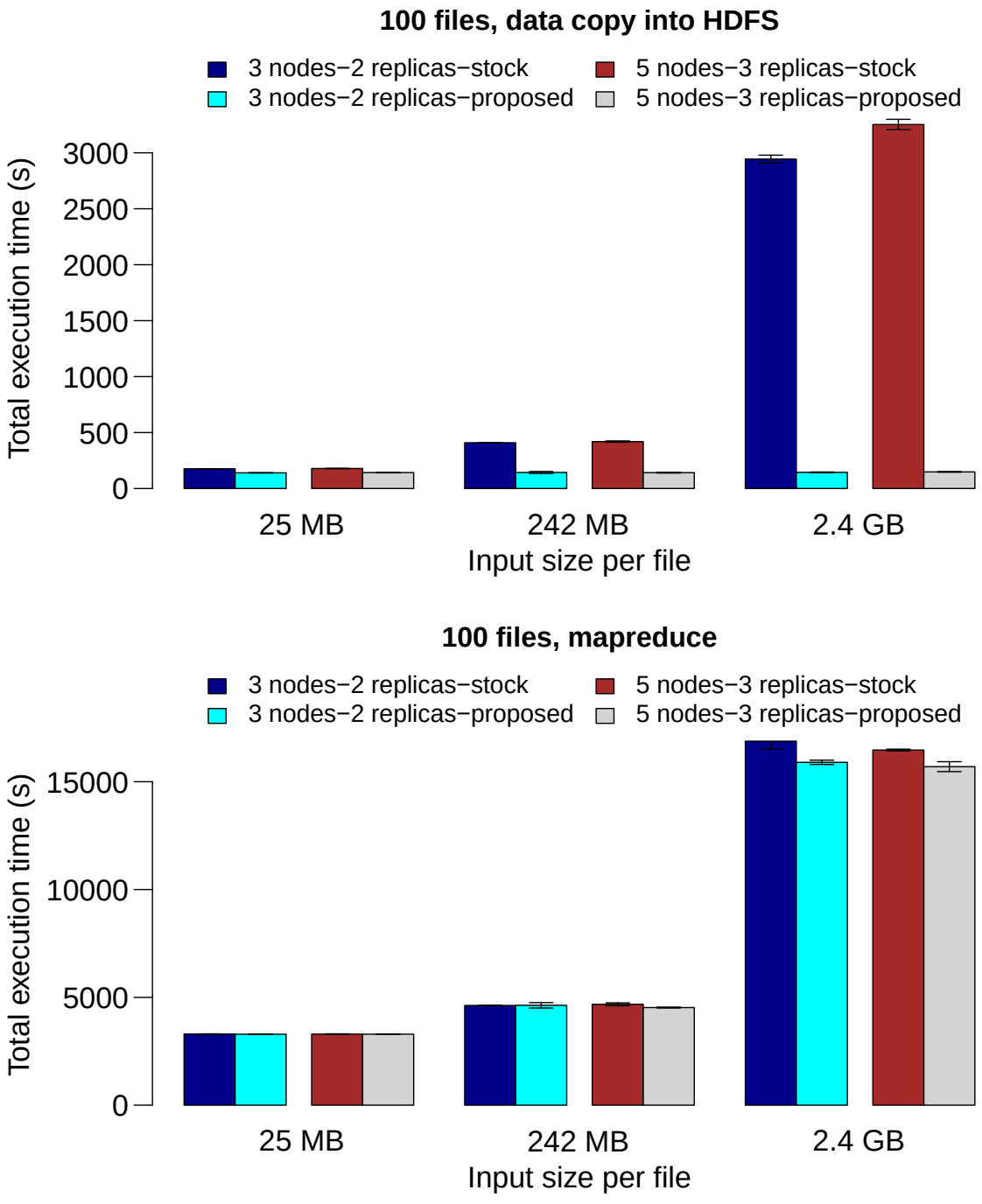


FIGURE 5.4.1: Evaluation results for *Grep*

timization is to the initial data copy phase, but as the input size per file becomes larger (i.e. 2.4 GB), we have nearly 5% of improvement in MapReduce performance. As we mentioned, although MapReduce computation performance is not a primary goal of this chapter, we were still able to achieve slight computational performance improvements due to data-compute locality. This is achieved by using the object storage system to identify replicas and force Hadoop to co-locate compute threads with the appropriate replica. Note, that there are a variety of existing techniques to co-locate data and compute, but none of these approaches help during the data copy phase. Similar to the data copy phase, the number of replicas or total storage nodes does not have a significant effect on the performance of the MapReduce phase.

5.4.2 Wordcount

We now present the experimental evaluation results for the *Wordcount* benchmark. This benchmark counts the number of occurrences of each word in a given file and extracts these numbers to the resulting output file, generating an output file that is of similar size to its input.

In our test scenario, we first generate 100 files that are equal to each other in size (29 MB, 286 MB or 2.8 GB). Similar to the *Grep* benchmark test in Section 5.4.1, stock Hadoop creates these files in HDFS and writes data to each. Also, in our proposed implementation, these files already exist in Ceph and we just make Hadoop aware of these existing files through symbolic links during its initialization. Following the data creation, we run the *Wordcount* benchmark on the newly created data. We perform the test steps above for both 3 nodes with 2 replicas and 5 nodes with 3 replicas.

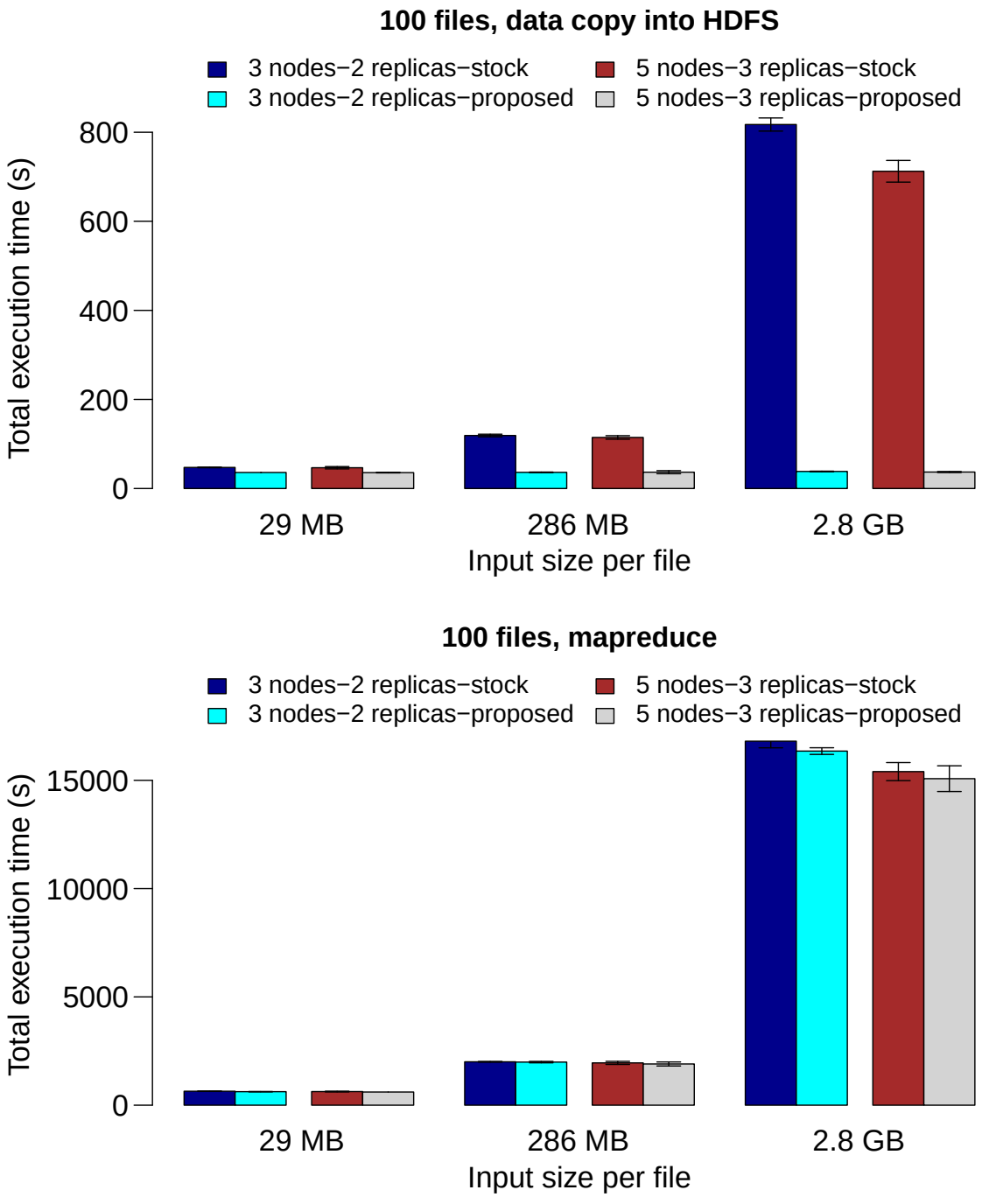


FIGURE 5.4.2: Evaluation results for *Wordcount*

We first measure the total time it takes to copy data into HDFS at first and show the results in the upper sub-plot of Figure 5.4.2. Similar to the results in Figure 5.4.1, our proposed implementation takes significantly less time than stock Hadoop to copy data into HDFS, as we are not ingesting any data into HDFS. As the input size per file is increased from 29 MB to 2.8 GB, the time our implementation spends to copy data into HDFS stays constant. Whereas, the time it takes for stock Hadoop to do the same increases. For *Wordcount* benchmark, we are able to achieve a 95% improvement in terms of initial data copy performance and as the input size becomes larger, this improvement rate will be much larger. Again, the number of replicas or total storage nodes in the system did not have a significant effect on the performance of the data copy phase. For the MapReduce phase, as the input size per file becomes larger (i.e. 2.8 GB), MapReduce performance improves as well (by nearly 5%) due to data-compute locality, with the number of replicas or storage nodes having no significant effect.

5.4.3 TestDFSIO

In this section, we present the experimental evaluation results for the *TestDFSIO* benchmark. This benchmark generates its own input with its *write* option using the MapReduce framework rather than the traditional data ingest method. It is possible to specify the number and size of files to generate with *TestDFSIO* benchmark. In our case, *TestDFSIO* does not generate any input data, because its input already exists in the system. When a *TestDFSIO* run is completed, it dumps statistics about the benchmark performance (throughput, execution time, io rate etc.)

For the sake of simplicity and as the number of input files will not have a significant

effect on our *TestDFSIO* tests, we tested the *TestDFSIO* benchmark with a single file and varied the file size between 500 MB and 50000 MB. Stock Hadoop creates these files with the *write* option of *TestDFSIO* and then performs a read operation on them. In our proposed implementation a zero-length write is performed that sets up symbolic links to already existing data, followed by a read operation. We perform these test steps for both 3 nodes with 2 replicas and 5 nodes with 3 replicas.

Figure 5.4.3 shows the data copy and MapReduce performance of our proposed implementation compared with that of stock Hadoop. We can first observe that, regardless of the input data size and the number of replicas and storage nodes, our implementation spends the same amount of time for the initial ingestion of data. On the other hand, the time it takes for stock Hadoop to create the data with the *write* option of *TestDFSIO* increases as the file size is increased from 500 MB to 50000 MB. At 50000 MB, our implementation achieves a 96% improvement over the stock Hadoop implementation in terms of data copy performance. For the MapReduce phase, as the file size is increased, we can see that our implementation performs better when compared to stock Hadoop. Since we are using local map tasks, in other words co-locate computation and data, the time it takes to shuffle mapper outputs to reducers is much less in our case. Additionally, *TestDFSIO* MapReduce phase is dominated by reading the generated data which happens totally local in our case, making the MapReduce improvement more apparent. As a result, we can see that our implementation improves MapReduce performance by nearly 20% with number of replicas or storage nodes having no significant effect. *TestDFSIO* test results are highly variant and this is a known issue with the *TestDFSIO* benchmark [15].

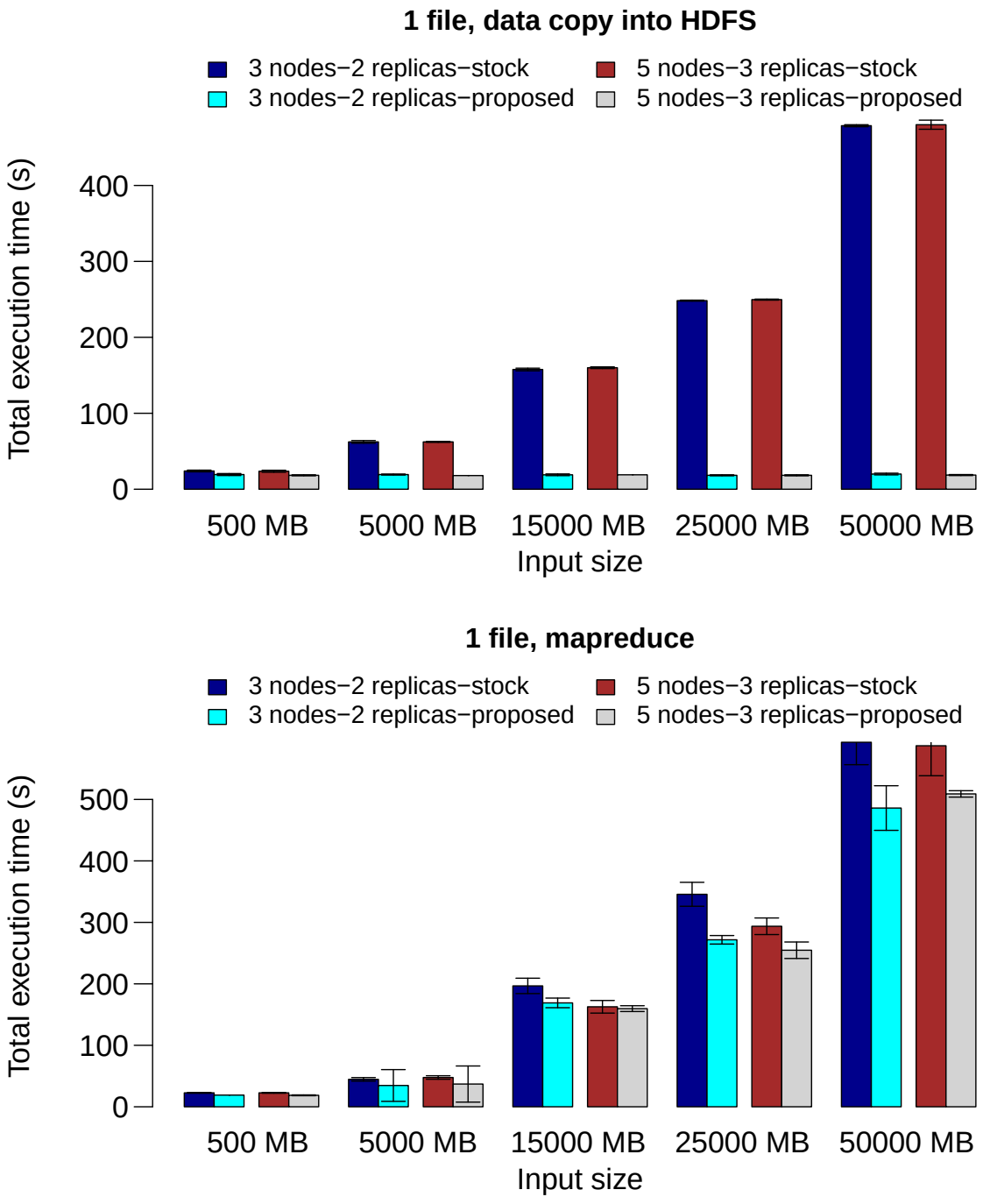


FIGURE 5.4.3: Evaluation results for *TestDFSIO*

5.4.4 TeraSort

We finally present the experimental evaluation results for the *TeraSort* benchmark. Similar to the *TestDFSIO* benchmark, *TeraSort* generates its own input with the *TeraGen* benchmark using the MapReduce framework rather than the traditional data ingest method. The input data generated by *TeraSort* consists of 100-byte rows. It is possible to specify the size of input data to be generated with *TeraGen*. In our proposed case, *TeraSort* does not generate any input data as its input already exists in the system.

We varied the input size for *TeraSort* benchmark between 5 GB, 10 GB and 50 GB during the performance evaluations. Stock Hadoop implementation creates files of these sizes with the *TeraGen* benchmark and then sorts them with *TeraSort*. Our implementation performs zero-length write when *TeraGen* is executed and creates symbolic links to already existing data. *TeraSort* sorts generated data and finally *TeraValidate* is executed to validate the sorted data for both stock Hadoop and our proposed implementation. This test case is performed for both 3 nodes with 2 replicas and 5 nodes with 3 replicas.

Data copy and MapReduce performance evaluation results of the stock Hadoop and our proposed implementation are shown in Figure 5.4.4. In the upper sub-plot of Figure 5.4.4, we observe that data copy times increase as the input size is increased from 5 GB to 50 GB for both stock Hadoop and our proposed implementation. We expect this for stock Hadoop. However, in our case this happens since we are still going through the records created by the *TeraGen* one-by-one. *TeraGen* generates data in the format of 100-byte records (i.e. to generate 6553600 bytes of data, 65536 records are needed) and it writes data to each record individually. We go through these

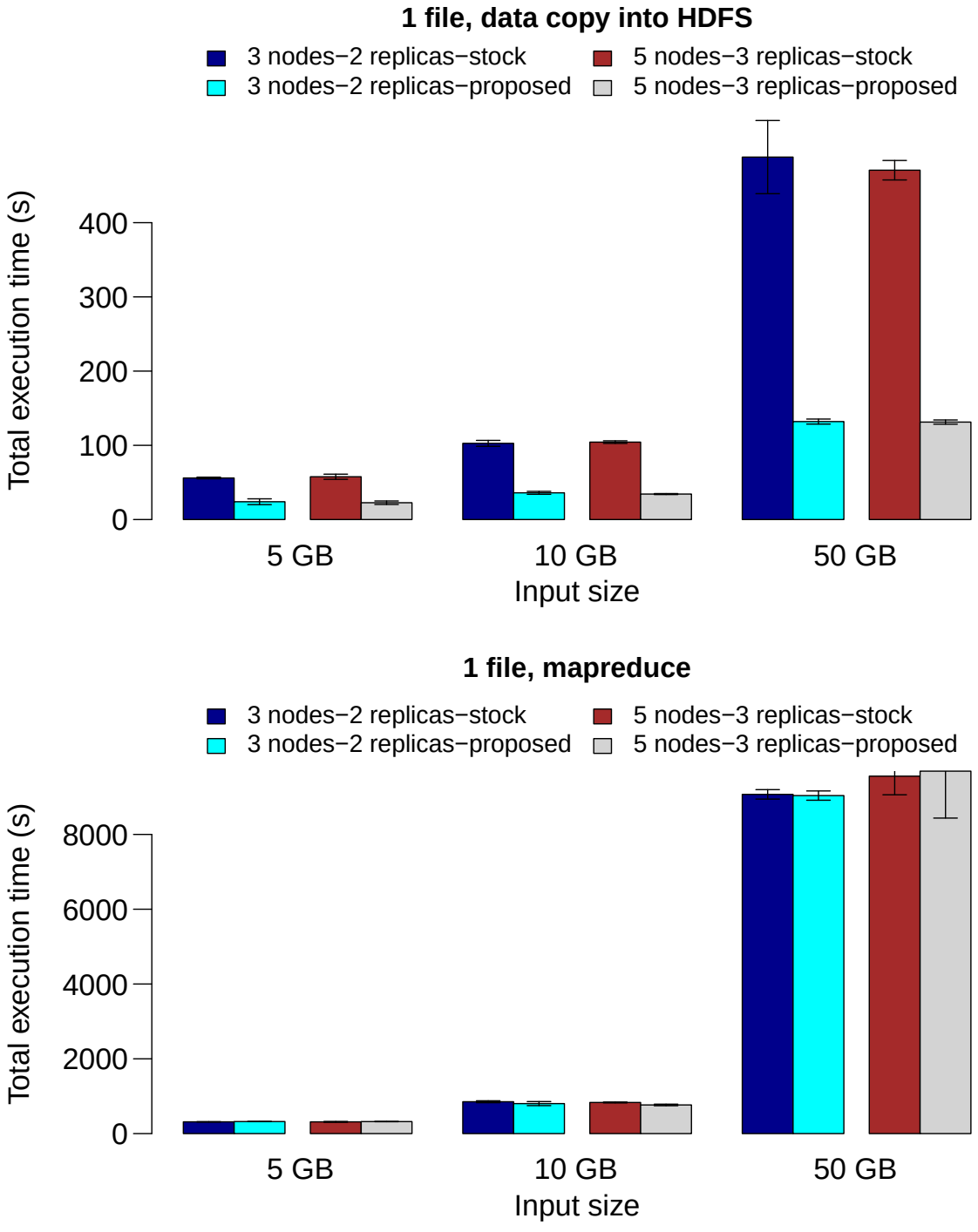


FIGURE 5.4.4: Evaluation results for *TeraSort*

records, but don't commit the actual data to HDFS by creating empty records. As the input size increases from 5 GB to 50 GB, number of records we go through increases as well which in turn causes data copy time to go up. Still, our implementation improves the data copy performance by up to 73% and as the input size becomes larger, we perform better with regards to data copy performance when compared against stock Hadoop implementation. Additionally, number of replicas and storage nodes does not affect the data copy performance significantly. For the MapReduce phase, we see nearly 5% improvement due to data-compute locality, with number of replicas or storage nodes having no significant effect. MapReduce improvements are not as high as those from *TestDFSIO*, as the MapReduce phase is not dominated by reads only.

5.5 Summary

In this chapter of the thesis, we presented an approach that performs computation on existing large-scale data in an object storage system without moving data anywhere and analyzed the outcomes of our approach. Experimental evaluations with Hadoop and the Ceph object-based storage system show that it is possible to implement Hadoop on top of Ceph as a lightweight computation layer and to perform computational tasks in-place alleviating the need to transfer large-scale data to a remote compute cluster. We have seen up to 96% of improvement in the initial data copy performance and up to 20% of improvement in the MapReduce performance.

Chapter 6

Parity-based Redundancy on Object Storage

Distributed storage system designers have traditionally emphasized maximizing I/O performance and minimizing cost. However, data reliability is a very important requirement of most storage systems. Reliability measures are often eliminated from the storage system design due to additional performance and cost overheads, particularly in high-performance computing systems. However, as mentioned in Chapter 1.1, there might be use cases where data reliability is a requirement. Replication is the most common technique to achieve reliability, but it is expensive in terms of storage space. For such systems, an alternative approach is to use parity-based redundancy techniques that utilize much less additional storage space. However, these techniques rely on clients and file systems to calculate and transfer parity within the storage system and do not take advantage of the existing object storage intelligence at all.

In this chapter, we propose a parity-based redundancy scheme which is implemented completely on object storage. We manage parity generation and writes at

the storage node instead of the client or file system level. Moving redundancy management to the storage level reduces the cost of data and parity rewrites and makes data management more efficient due to the features of the object storage. It also enables developing a portable object storage framework that can easily be interfaced with different clients and file systems. Additionally, object storage offers additional features for efficient data organization, such as grouping objects in collections, that can optimize data reconstruction during failures, based on client needs.

The rest of this chapter is organized as follows: Section 6.1 introduces related studies in the area. Section 6.2 describes key design approaches, along with the specific design goals and details of our implementation. Section 6.3 presents the results of experimental evaluations and we conclude this chapter in Section 6.4.

6.1 Related Work

This section highlights some of the existing work on parity-based redundancy techniques. Narayan et al. developed a series of algorithms for a fault-tolerant distributed storage system [95]. The proposed algorithms preserve data and file system integrity and consistency in the presence of concurrent reads and writes. This work paired the Lustre [40] file system layer with an object storage architecture, while trying to minimize its effect on the overall system performance. However, data was stored locally both at the client and the cluster to reduce the impact of parity generation on writes via a dirty region database (DRD) and this increases storage space overhead. Similarly, Zebra is a network file system that increases throughput by striping file data across multiple servers [71]. Zebra attempts to tackle some of the RAID-induced

bottlenecks by striping large writes (to take advantage of parallelism), aggregating small writes via logging and performing large transfers and creating multiple paths between the source of data and the disks, so that different paths can be used to reach different disks. Like other implementations, here the rewriting or recalculation of parity is done at a single point by reading back all the required old data and then using the new data to calculate parity.

The Panasas ActiveStor File System uses parallel and redundant access to object storage devices (OSDs) in addition to per-file RAID. It is able to provide scalable performance to many concurrent file system clients through parallel access to file data that is striped across the storage nodes. This is possible due to the clustered design of the storage system and the use of client-driven RAID. However, client-driven RAID [131] requires clients to be responsible for computing and writing the parity, along with incurring further communication costs.

Reisner et al. developed a series of algorithms for dealing with the issues of shared resource networks (e.g. split-brain) [107]. The proposed approach, DRBD, works in a fashion similar to RAID1 by creating identically mirrored copies of the previously shared data. Like our work, DRBD does not require changes to the client parallel file system, but DRBD does not support higher level RAID parity (e.g. RAID4, RAID5, etc.) implementations.

Finally, the SCSI command set contains a model for XOR-based operations [65], supervised by an array controller. The basic XOR commands of interest are: *XDWRITE*, *XDREAD* and *XPWRITE*. The parity functions that we propose perform tasks similar to the functions above. However, the SCSI model requires an array controller to supervise the XOR operations and as a result, the array controller can be thought of a hardware-level client, similar in concept to the previous cases described

in this section.

6.2 Design Approach

This section introduces the implementation of parity-based redundancy on object storage to improve data reliability and availability. In particular, object storage has been used as storage back-end and data and parity management tasks have been offloaded to the storage units. In a subsequent section, we show how such an implementation reduces network hops and thereby improves performance. Using parity-based RAID [87], we facilitate safekeeping of data without having to fully replicate it, thereby saving storage resources. We have implemented our approach using OSC-OSD emulation [54] and PVFS [45].

Following section gives a brief overview of Redundant Array of Inexpensive Disks. We refer the reader to Section 2.1 for an overview of object storage.

6.2.1 Fundamental Design Elements

RAID

Redundant Array of Inexpensive Disks (RAID) refers to a set of techniques that use additional cheap disk space to provide data reliability [101]. The basic form of RAID, referred to as $RAID1$ which is popular among large-scale systems is replication or keeping one or more duplicates of data. While this may be desirable for large-scale systems such as the storage system back-ends of social media websites, more cost efficient techniques involve $RAID4$ and $RAID5$ which use $1/(n)$th additional disk

space. In $RAID4$, a dedicated disk contains parity stripes corresponding to actual data stripes in the remaining disks. Parity stripes are calculated by bitwise XORing the data stripes from the remaining disks. $RAID5$ improves on the single disk parity bottleneck by distributing parity over each of the disks. In doing so, $RAID5$ reduces the workload on a single disk in performing parity calculations and maintenance while still using the same amount of additional disk space. In this chapter, we introduce the implementation of both $RAID4$ and $RAID5$ parity schemes on object storage devices (OSDs).

6.2.2 Implementation Details

Communication between OSDs

The current implementation of OSDs in T10 [124] does not allow a communication request to be generated at one OSD target and to be transferred to another one. However, clients of OSC-OSD emulation [28] can send commands to the OSDs using the *osd-initiator* library and OSDs can reply back to the clients with corresponding outputs. As an example, a PVFS client can send a *create* command to an OSD using the *osd-initiator* library. The *osd-initiator* library has been developed to translate requests from a file system client into OSD commands and to transfer OSD responses back to the client. After receiving the command, OSD creates an object and replies back to the client with the identifier of the newly created object as shown in Figure 6.2.1.

In order to enable communication between the OSDs, we have integrated the *osd-initiator* library into the OSD software emulation [54], so that OSDs can initiate communication with other OSDs and send or receive commands to or from other

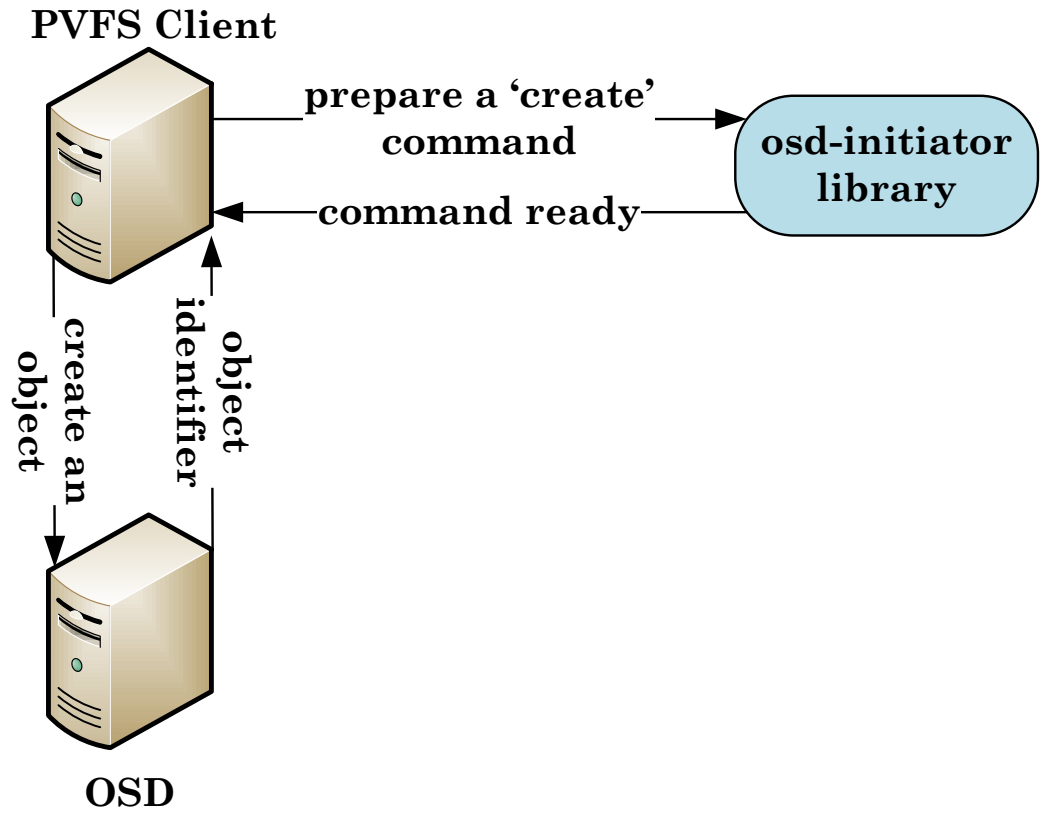


FIGURE 6.2.1: An example of a PVFS client creating an object on an OSD using *osd-initiator* library

OSDs as shown in Figure 6.2.2. Following this approach, each OSD will be aware of other OSDs in the system. As a result, if the system consists of n OSDs, then there will be $n * (n - 1)$ communication links between these OSDs.

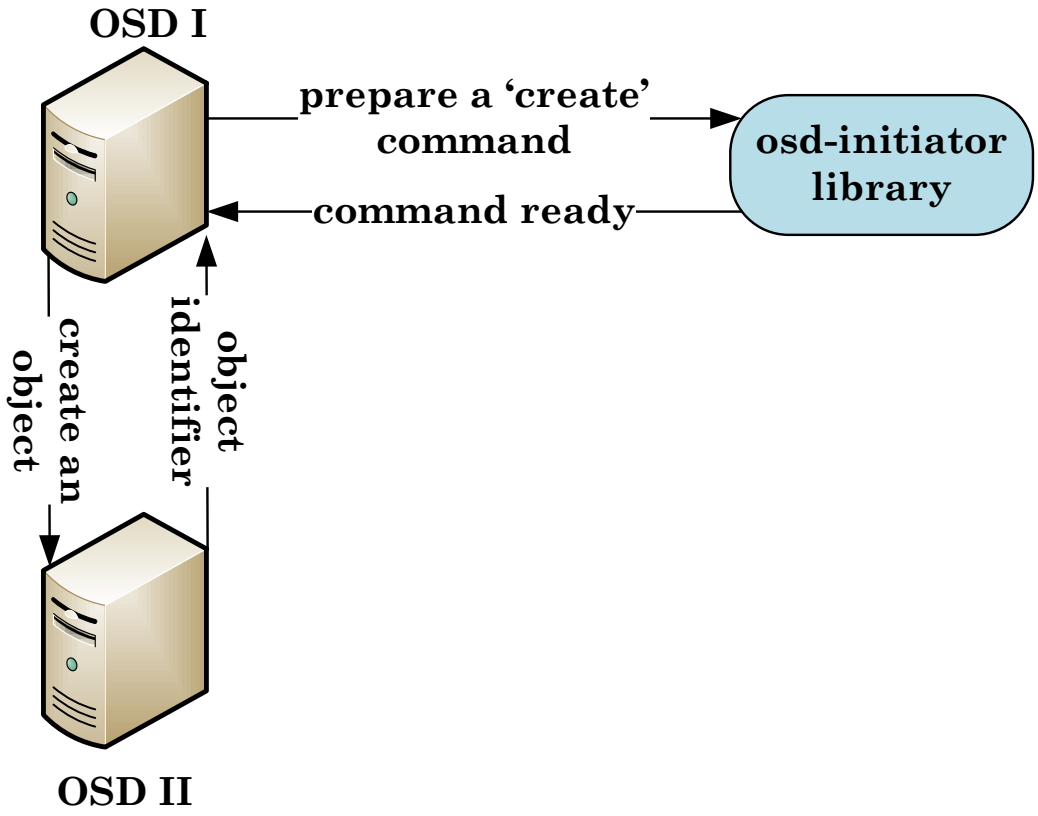


FIGURE 6.2.2: An example of an OSD creating an object on another OSD using *osd-initiator* library

Client-level versus Storage-level Parity

In this section, we distinguish the traditional approach of redundancy measure, i.e, client-level parity handling, from our approach of moving parity handling to the stor-

age units. Client-level parity scheme requires that a client takes charge of calculating the parity and writing new data and parity to the correct locations. In case of small writes, a client will follow the procedure below that is also shown in Figure 6.2.3, assuming that the client writes data to OSD-II.

- Read the old data
- Read the old parity value
- Calculate the intermediate parity value
- Write the new data to the correct OSD
- Write the new parity to the parity OSD

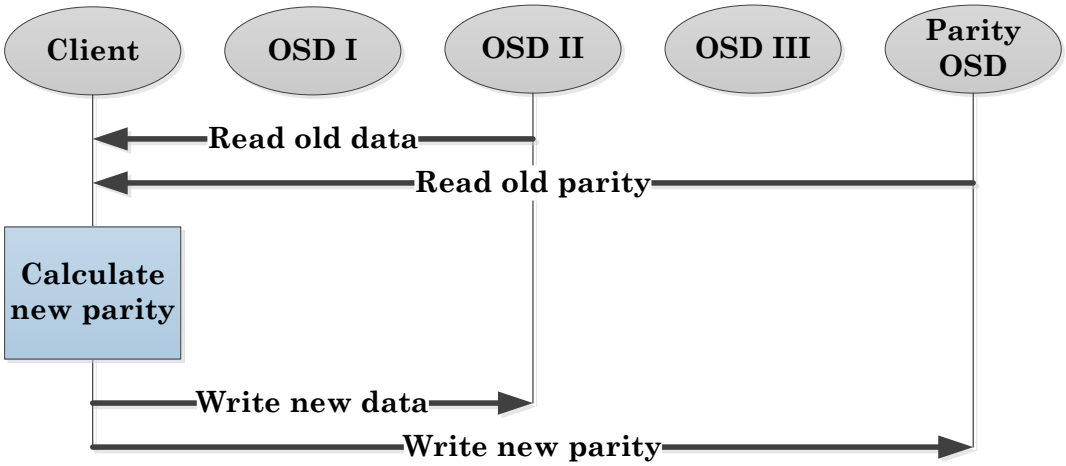


FIGURE 6.2.3: Client-level parity handling

Storage-level parity, as we are proposing, however, simplifies this series of communications by having the client follow the procedure below that is also shown in Figure 6.2.4, assuming that the client writes data to OSD-II.

- Send the new data to the correct OSD
- Target OSD calculates the intermediate parity value by using the old data
- Target OSD writes the new data to itself
- Target OSD forwards the parity value to the parity OSD
- Parity OSD calculates the new parity value by using the old and forwarded parity values
- Parity OSD writes the new parity to itself

In this manner, storage-level parity is able to eliminate two data communication steps that would normally occur in the client-level implementation. While it is true that this cost reduction is non-existent in the large write scenario (since it would be easier for the client to simply perform all the calculations and then simply write all the data at once), the reduction in communication overhead still plays a significant role for small writes. It is also important to note that, the parity calculation and update operations are completely transparent to the client, meaning that the client does not need to know if there is any redundancy mechanism implemented in the storage system.

Parity OSD Assignment

As we have mentioned earlier, we have implemented *RAID4* and *RAID5* parity schemes on top of OSDs. At this point, it is useful to discuss how a parity group over the OSDs is defined and how the identifiers of the objects in a parity group are related to each other.

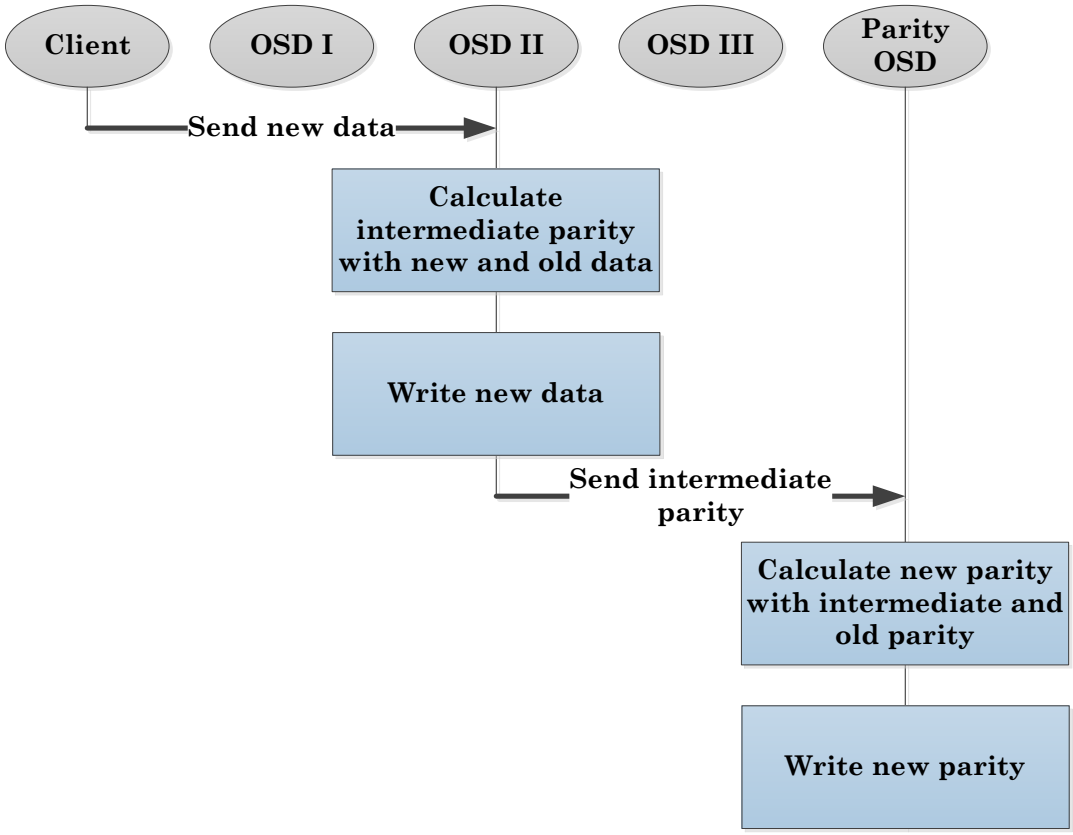


FIGURE 6.2.4: Proposed storage-level parity handling

First of all, we assume that in a group of n OSDs, each OSD has a unique identifier range as shown in Figure 6.2.5. This might not be the case for all the clients built on OSDs, but OSDs can be easily configured to have unique identifier ranges.

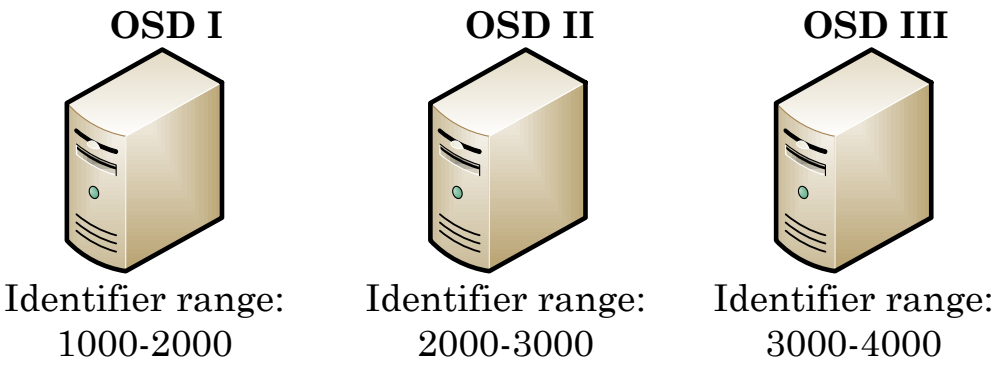


FIGURE 6.2.5: An example of identifier ranges of OSDs in proposed storage-level parity handling

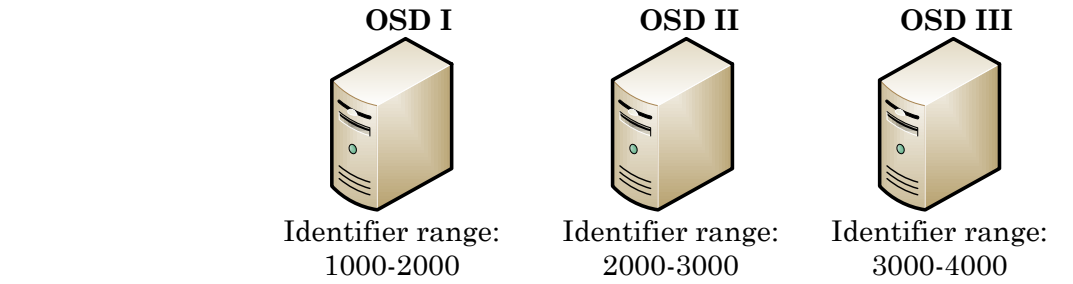
We define a *parity group* as a group of objects that have identifiers equally larger than the smallest identifier on an OSD. This means that if the smallest identifiers of the OSDs shown in Figure 6.2.5 are x, y and z, then identifiers $x+a$, $y+a$ and $z+a$ belong to the same parity group, where a is the offset from the smallest identifier on each OSD. An example of parity group assignment can be seen in Figure 6.2.6.

Once the parity groups are assigned, the parity OSD can be identified depending on the parity scheme: *RAID4* or *RAID5*. In *RAID4*, one of the OSDs will be assigned as a parity OSD and will not be used by the client for regular I/O operations. This assignment can be done in a random fashion, meaning that in Figure 6.2.5, OSD-I, II or III can be chosen as the parity OSD. Parity OSD identification is a bit more complicated in *RAID5* parity scheme. In this configuration, the parity will be distributed across the OSDs. Assume we have n OSDs and they know their

Parity Group 0	1000	2000	3000
Parity Group 1	1001	2001	3001
Parity Group n	1000+n	2000+n	3000+n

FIGURE 6.2.6: An example of parity groups on OSDs in proposed storage-level parity handling

position in the OSD array during initialization time, meaning that OSD at position m knows that there are $n-1$ other OSDs available. Once all OSDs are started successfully during the initialization time, they reserve certain identifiers for *RAID5* parity objects only. For an OSD at position m that knows there are $n-1$ OSDs in the system other than itself, the reserved *RAID5* parity identifiers should satisfy the equation $((identifier \bmod smallest\_identifier) \bmod n = m-1)$. An example of reserving *RAID5* parity objects with the OSDs in Figure 6.2.5 is shown in Figure 6.2.7.



Parity Group 0	**1000**	2000	3000
Parity Group 5	1005	2005	**3005**
Parity Group 10	1010	**2010**	3010

FIGURE 6.2.7: An example of reserving RAID-5 parity objects on OSDs in proposed storage-level parity handling (numbers in bold are reserved RAID-5 parity identifiers)

Reserved *RAID-5* parity objects cannot be used for regular I/O operations. We modified each OSD command with a flag indicating if they are parity create or update commands. If an OSD finds out that an incoming OSD command, whether it is from a client or another OSD, does not have the parity flag set, then it will not allow that

client or OSD to use the reserved *RAID-5* parity object identifiers. The reserved identifiers may only be used for newly created or updated parity objects.

Parity Creation and Update

Now that we mentioned how to identify the parity OSD in *RAID4* and *RAID5* configurations, it is a good point to discuss how the parity objects are created or updated in both parity schemes in our implementation.

In our implementation, any read or write operation initiated by the client causes the creation or update of a parity object in the parity OSD. Initially the OSDs will not store any object. In this case, if a *create and write* request with data buffer *newdata* from the client arrives at an OSD, the OSD will create a data object and write to it right away and then based on the identifier of the newly created and written object, a *write* command will be forwarded to the parity OSD. If the identifier of the newly created and written object is $x+a$, where x is the smallest identifier on that OSD and a is the offset from the smallest identifier, then the *write* command being transmitted to the parity OSD will use identifier $z+a$, where z is the smallest identifier on the parity OSD and a is the offset from the smallest identifier. Once the parity OSD receives the *write* command for an object with identifier $z+a$, it will first check the existence of this object. If it does not exist, that means the parity object has not been created yet. So, a *create and write* command with the data buffer *newdata* will be called by the parity OSD. Otherwise, it means the parity object has already been created and written to. Therefore, an XOR operation will be performed with the existing parity and data buffer *newdata* to update the existing parity object. The process of creating or updating a parity object is illustrated in Figure 6.2.8, assuming

that the initial *create and write* command by the client is served by OSD-II.

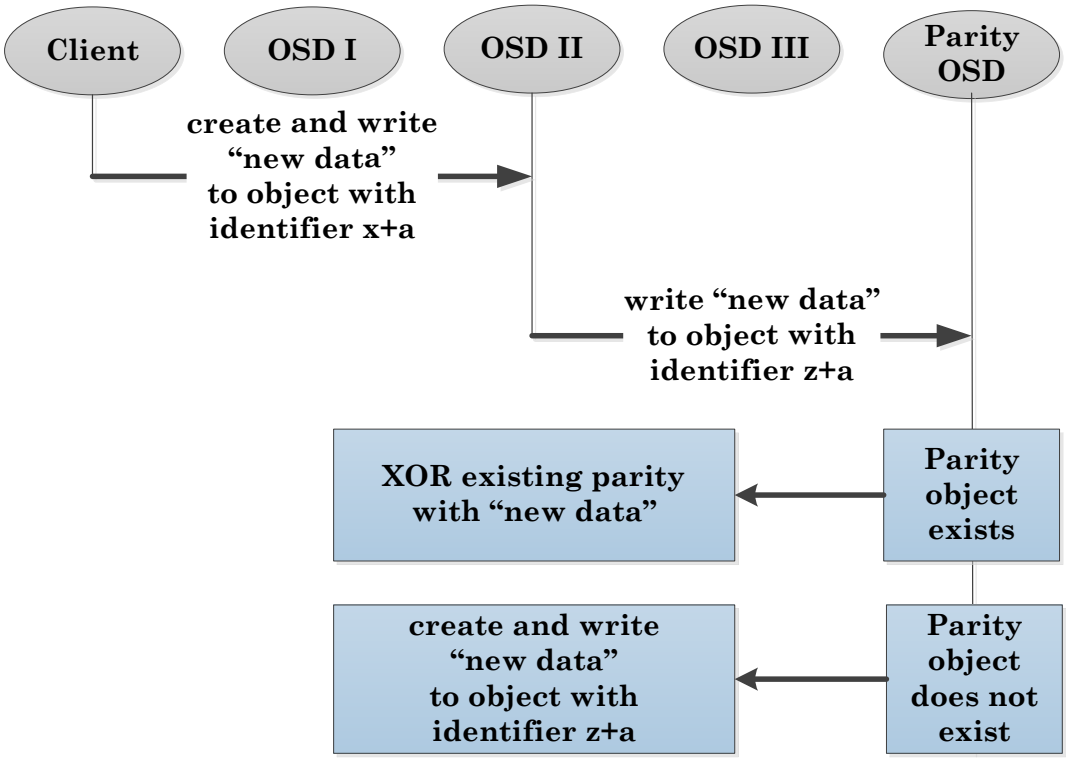


FIGURE 6.2.8: An example of creating or updating a parity object on an OSD in proposed storage-level parity handling

If the client writes to an existing object with an identifier $x + a$, rather than creating and writing to a new object, then an XOR operation needs to be performed with the existing data and the new data of the object to form $forwardparity$ locally. Then the result of this XOR operation, $forwardparity$, is forwarded to the parity OSD with a *write* command using the identifier $z + a$ as in the previous example. Finally, the parity OSD will check for the existence of the parity object as shown in Figure 6.2.8. If the parity object does not exist, it will create and write $forwardparity$ to it. Otherwise, it will perform XOR between the existing parity and $forwardparity$

to update the existing parity object. The process of creating or updating a new parity object when the client overwrites an existing file is illustrated in Figure 6.2.9, assuming that the initial *write* command by the client is served by OSD-II.

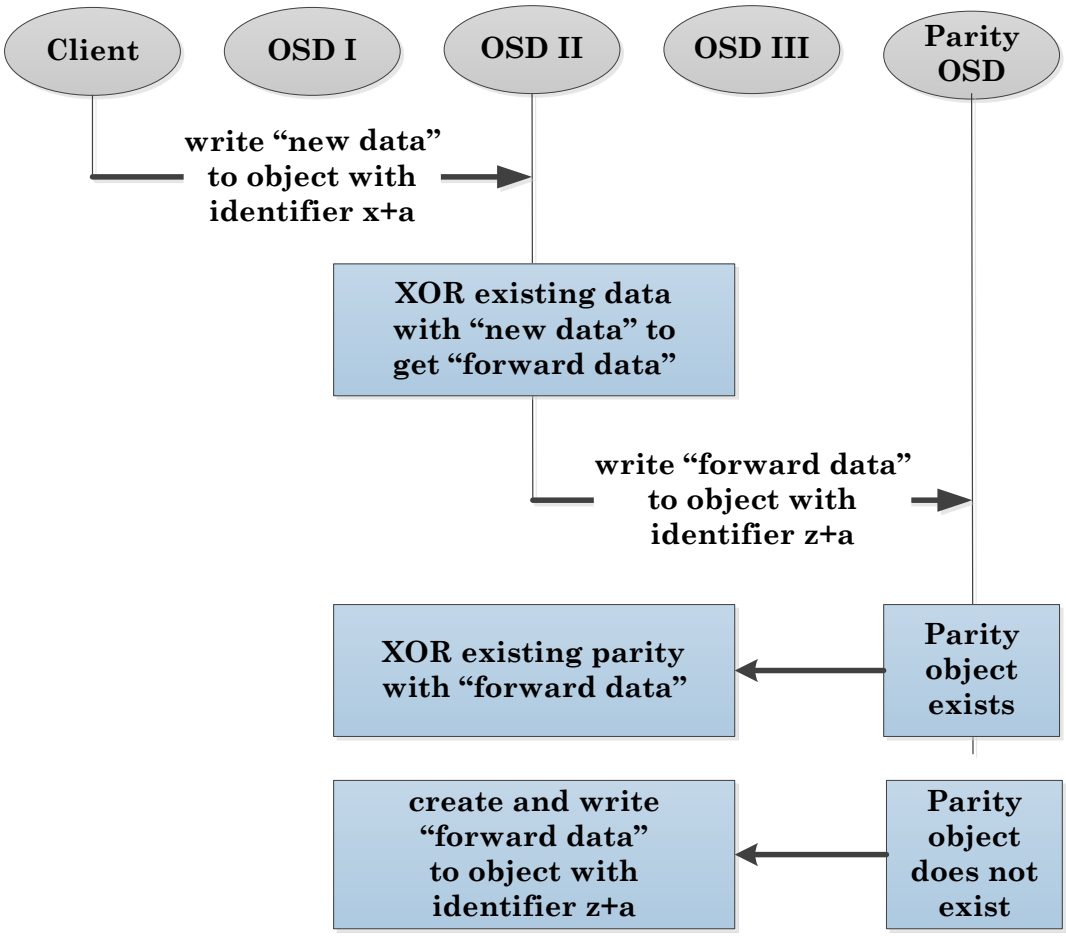


FIGURE 6.2.9: Another example of creating or updating a parity object on an OSD in proposed storage-level parity handling

As can be inferred from Figures 6.2.8 and 6.2.9, the total number of communication steps is only two - first, one between the client and the OSD; and second, one between the OSD and the parity OSD. Since we assume performing XOR operations inside

the OSDs locally will be significantly fast, we expect the overhead of parity create or update operations to be reasonably small.

Data Reconstruction

Data reconstruction comes into play when an OSD (or a subset of objects on an OSD) fails. At this point the missing data has to be regenerated from the existing data. The basic operation in reconstruction is the regeneration of a single object. A single object can be regenerated through bitwise XORing of the remaining items in a parity group and the parity object for that group. We have equipped each OSD with the capability of fetching objects from other OSDs and serially XORing them to reproduce missing data. Figure 6.2.10 below illustrates the process of a single object reconstruction.

In Figure 6.2.10, the client wants to read object $w + a$ which it knows is stored locally on an OSD that is currently unavailable. When the client's network request times out, it can re-issue its access request to one of the active OSDs. Because we want portability and transparency, the client needs to only reissue the same read request without any other modification. We have extended the OSD_READ command on the OSD targets to handle the re-directed request by recognizing that the associated object identifier is outside of the identifier range of the given OSD. When the OSD encounters an out of range identifier, it extracts the offset, a from the identifier and queries the remaining active OSDs including the OSD on which the parity object resides. For each query, the OSD uses the offset and the start index of the identifier range of the queried OSD to determine the required object identifier to query. The results of the queries are serially XORed to finally generate the missing object. The

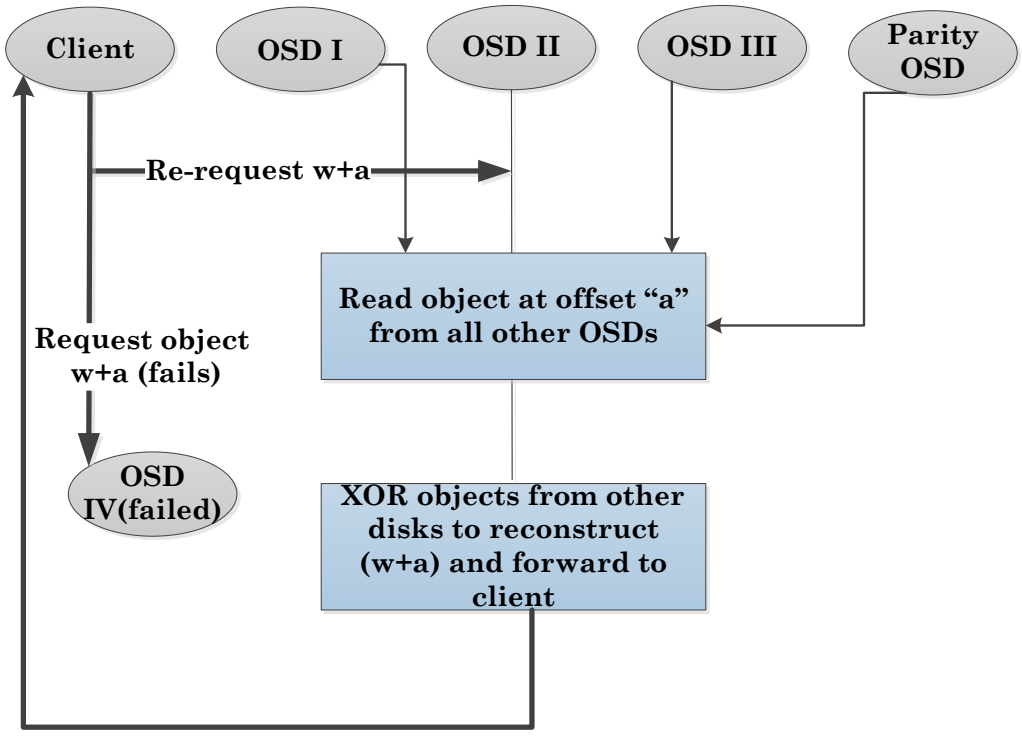


FIGURE 6.2.10: Reconstructing an object per client request

resulting object is then sent back to the client. The key concept here is that the client does not need to issue a special request to read a missing object.

Single object reconstruction can serve as building block to a more advanced regeneration functionality where a whole OSD is regenerated. A complete OSD regeneration can be performed during periods of reduced I/O workload. Full OSD regeneration simply requires extending the basic object reconstruction to iteratively traverse the entire identifier range of the failed OSD. To provide this functionality we have appended the OSD command set with the *OSD_Reconstruction* command. A client issues this reconstruction command in the same manner as other OSD commands. Once issued, the OSD servicing the request performs the remainder of the task without further client interference.

6.3 Evaluation

Performance evaluations were conducted on an HPC cluster consisting of 16 64-bit nodes, each running Ubuntu 12.04 64-bit edition.

6.3.1 Parity Creation and Update

In order to learn the overhead of parity creation and update operation, the execution time of the stock OSD create function is compared with the execution time of the OSD create function supporting redundancy for both RAID4 and RAID5 schemes. The clients in this test are based on PVFS [45]. The number of clients has been varied between one and four, whereas, the number of OSDs supporting redundancy has been varied between two and twelve. The clients create 1000 files on the OSDs

and the size of each file has been varied between 1 KB and 100 MB. The total time create operations take is measured and divided by one thousand in order to obtain time per each create operation in a PVFS system based on OSD storage that supports redundancy.

RAID4 vs RAID5

The first test case compares the two parity based redundancy schemes we have implemented on OSDs - RAID4 and RAID5. A single PVFS client is used in this test case and it creates 1000 objects with sizes of each varying between 1 KB and 100 MB on two OSDs. In RAID4 parity based redundancy scheme, one of the OSDs is chosen as the parity OSD. So, RAID4 with two OSDs becomes essentially *replication*. In RAID5, though, the parity OSD is chosen depending on the identifier of the newly created object as described in Section 6.2.2.

Figure 6.3.1 shows the results of the test comparing the performance of RAID4 with that of RAID5. As can be inferred from Figure 6.3.1, RAID4 turns out to be a better redundancy scheme compared to RAID5 in terms of time per create. This fact is true for larger file sizes in particular. In RAID4, the parity OSD is fixed. Therefore, the parity OSD will not consume its resources for hosting regular objects. In RAID5 though, the parity objects can be located in any of the OSDs and this requires OSDs to handle both regular objects and parity objects together causing an overhead compared to RAID4. The performance advantage of RAID4 over RAID5 is only true for small numbers of OSDs though, since for large numbers of OSDs in the system, the single parity OSD in RAID4 becomes overburdened with requests from other OSDs. Therefore, through the rest of the performance evaluations, the RAID5

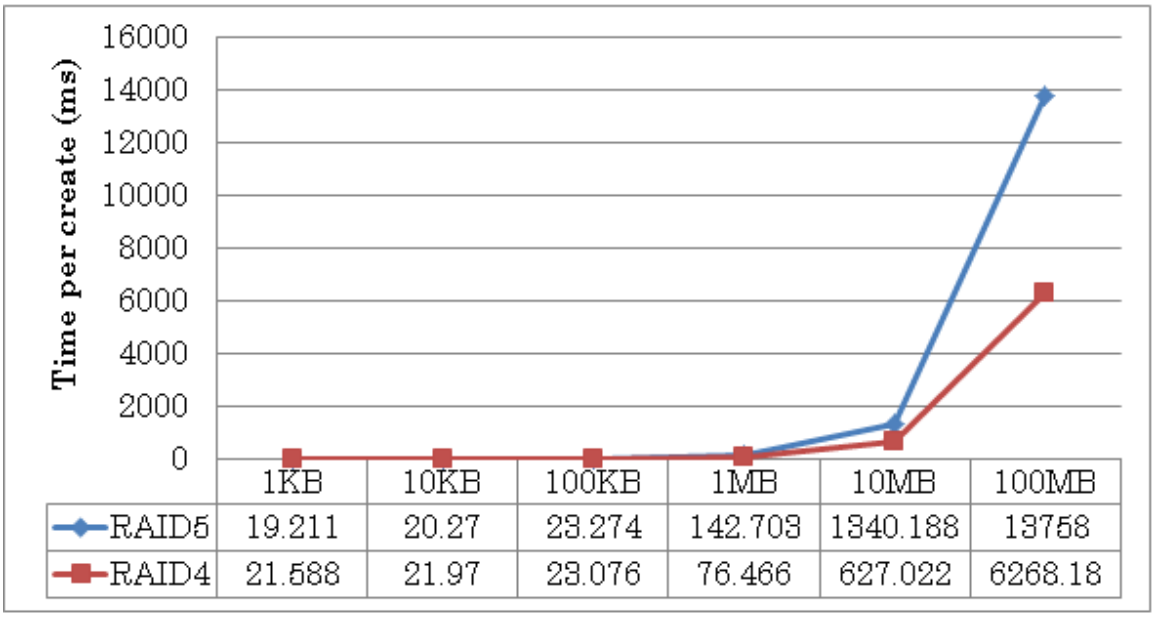


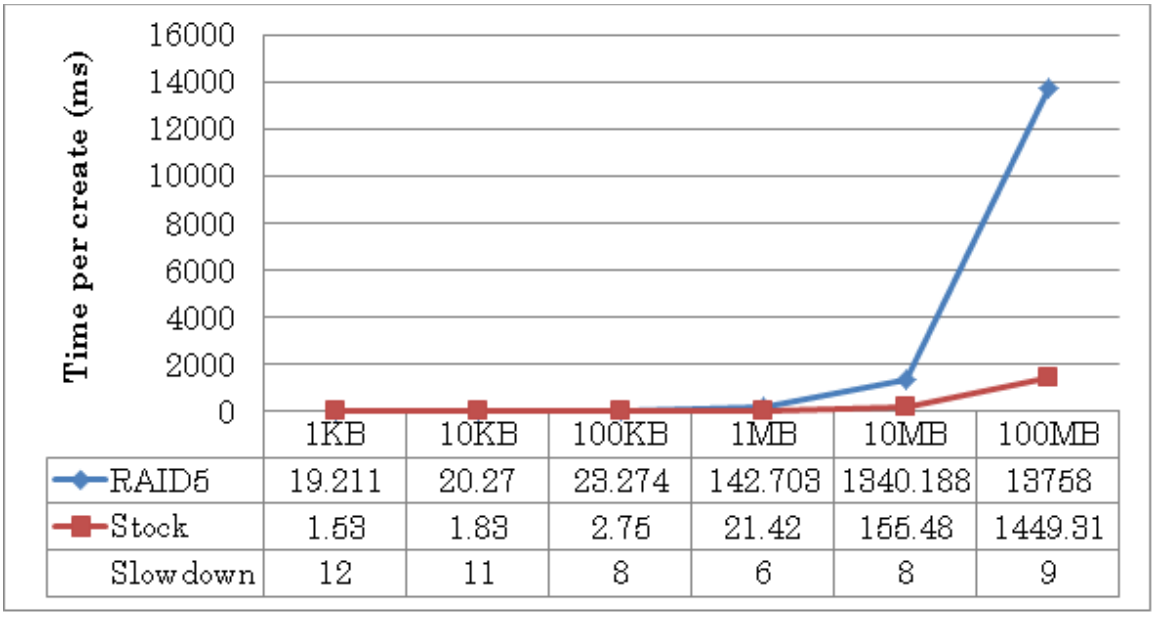
FIGURE 6.3.1: Time (in milliseconds) per creating a file of various sizes with RAID4 and RAID5 redundancy schemes (single PVFS client, 2 OSDs)

parity scheme has been implemented on OSDs.

Varying number of OSDs

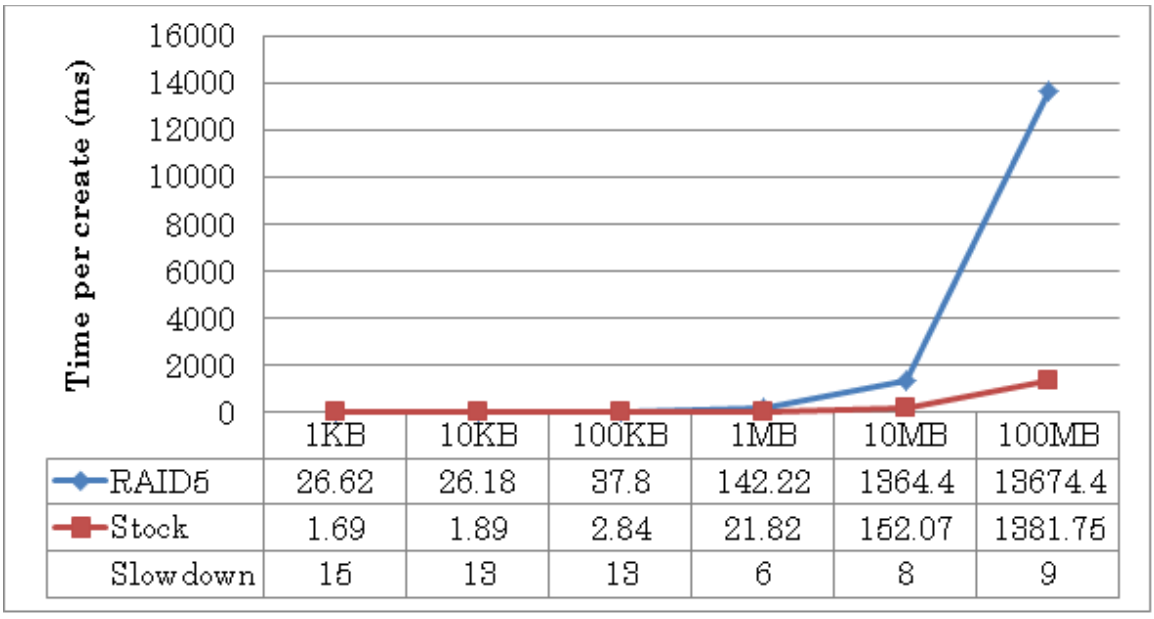
In the second test case, a single PVFS client creates 1000 objects with sizes of each varying between 1 KB and 100 MB on two, four and twelve OSDs supporting RAID5, in order to understand the effect of file size and the number of OSDs on the performance of the system. The performance metric here is again the duration of time a create operation takes for an object.

Figures 6.3.2, 6.3.3 and 6.3.4 show time per create for OSD implementation supporting RAID5 and stock OSD implementation. Time per create is shown for various file sizes (between 1 KB and 100 MB) with a single PVFS client and two, four and twelve OSDs in the system respectively.



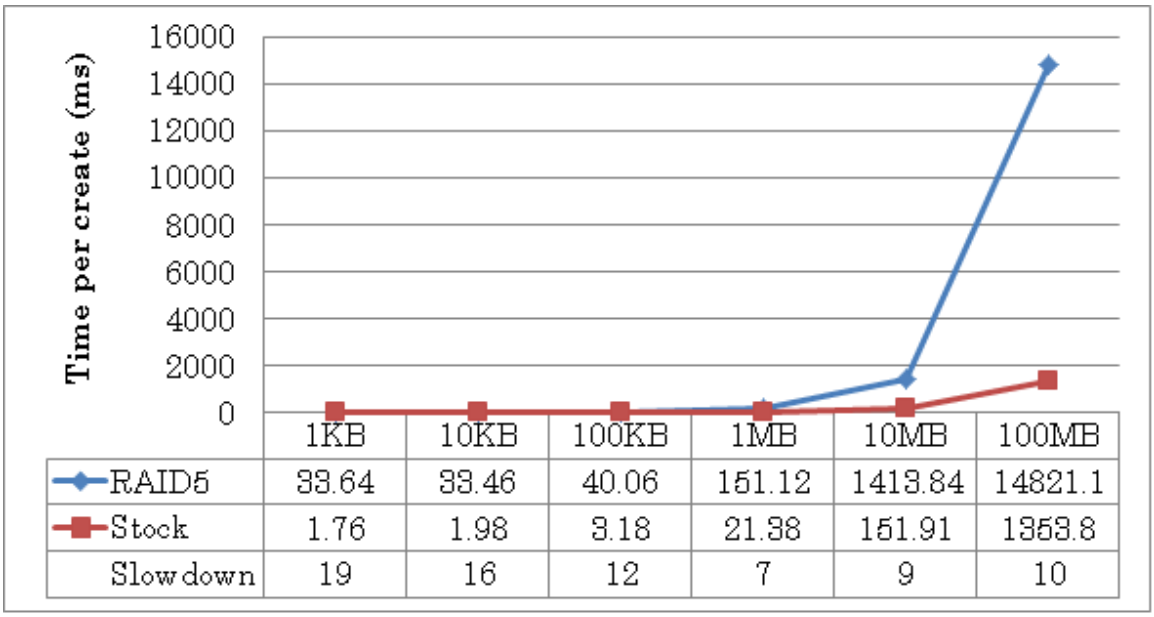
	1KB	10KB	100KB	1MB	10MB	100MB
RAID5	19.211	20.27	23.274	142.703	1340.188	13758
Stock	1.53	1.83	2.75	21.42	155.48	1449.31
Slowdown	12	11	8	6	8	9

FIGURE 6.3.2: Time (in milliseconds) per creating a file of various sizes with RAID5 redundancy scheme and stock PVFS-OSD implementation (single PVFS client, 2 OSDs)



	1KB	10KB	100KB	1MB	10MB	100MB
RAID5	26.62	26.18	37.8	142.22	1364.4	13674.4
Stock	1.69	1.89	2.84	21.82	152.07	1381.75
Slowdown	15	13	13	6	8	9

FIGURE 6.3.3: Time (in milliseconds) per creating a file of various sizes with RAID5 redundancy scheme and stock PVFS-OSD implementation (single PVFS client, 4 OSDs)



	1KB	10KB	100KB	1MB	10MB	100MB
RAID5	33.64	33.46	40.06	151.12	1413.84	14821.1
Stock	1.76	1.98	3.18	21.38	151.91	1353.8
Slowdown	19	16	12	7	9	10

FIGURE 6.3.4: Time (in milliseconds) per creating a file of various sizes with RAID5 redundancy scheme and stock PVFS-OSD implementation (single PVFS client, 12 OSDs)

As can be seen in these figures, the slowdown ratio compared to the stock case decreases as the file size is increased up to 1 MB and the slowdown ratio compared to the stock case increases as the file size is increased from 1 MB to 100 MB. The slowdown ratio compared to the stock case varies between 6 and 19. Previous work [95] showed a slowdown factor of about 20 for a naive implementation of client based redundancy.

Another interesting thing to note is that in all three figures, a file size of 1 MB seems to be the best in terms of overall performance while supporting redundancy at the same time. For file sizes less than 1 MB, the parity overhead is larger due to the overhead coming from creating small files using the PVFS client. By default, PVFS has a stripe size of 64 KB and when the file size is smaller than the default stripe size, data buffers are allocated and transferred over the network where only a small

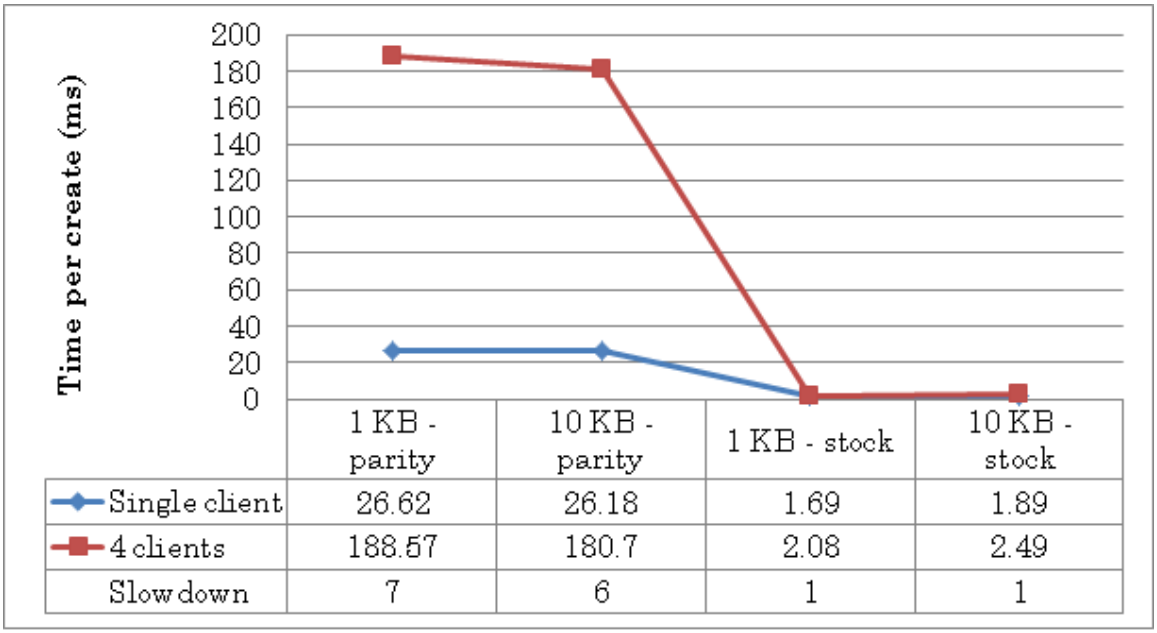
portion of these buffers is occupied. And as expected the parity overhead increases as the file size is increased from 1 MB to 100 MB and this is due to the overhead coming from local XOR operations and transferring larger chunks of data across the communication network.

One last thing to point out about Figures 6.3.2, 6.3.3 and 6.3.4 is that, time per create values for a single client with various number of OSDs are about the same. Therefore, we can say that, for a single client, the number of OSDs in the system does not have that much of an effect on the file creation throughput.

Varying number of clients

Another interesting issue to consider is the number of clients performing parity creations or updates in the system. For this case, we have compared the time per create values of a single client with the time per create values of four clients by using four, eight and twelve OSDs supporting RAID5 redundancy scheme. Each client creates 1000 1 KB and 10 KB objects on OSDs simultaneously and we measure the time per regular create operation and create operation with RAID5 redundancy support and give the slowdown ratio between these two. Figures 6.3.5, 6.3.6 and 6.3.7 show time per create values for this test case.

Figures 6.3.5, 6.3.6 and 6.3.7 show that, for the stock case increasing the number of clients have negligible effect on the time per create values. However, for the case where OSDs support parity based redundancy, as the number of clients increase there is a slowdown ratio between four and seven. This is expected, since having more clients require more local parity calculations and increased traffic between OSDs. It is important to note that the slowdown ratio is the least when the number of OSDs



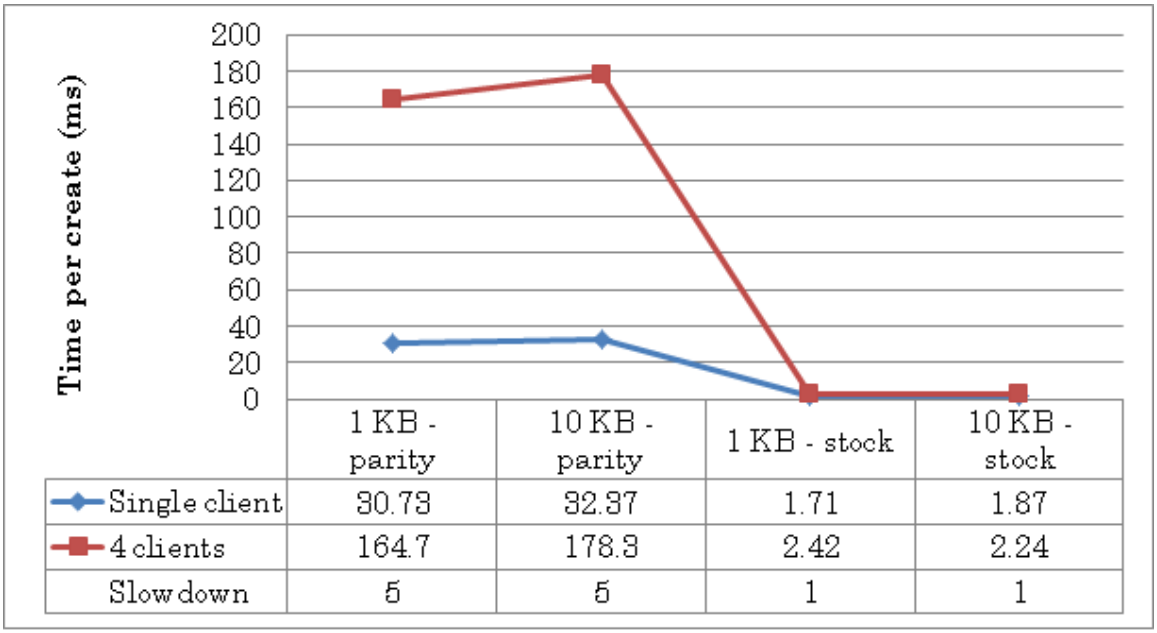
	1 KB - parity	10 KB - parity	1 KB - stock	10 KB - stock
Single client	26.62	26.18	1.69	1.89
4 clients	188.57	180.7	2.08	2.49
Slow down	7	6	1	1

FIGURE 6.3.5: Time (in milliseconds) per creating a file of various sizes and number of clients with RAID5 redundancy scheme and stock PVFS-OSD implementation (single and 4 PVFS clients, 4 OSDs)

in the system is twelve. As the number of OSDs in the system goes up, then clients have more resources for their concurrent parity create and update operations. Thus, the slowdown ratio goes down.

6.4 Summary

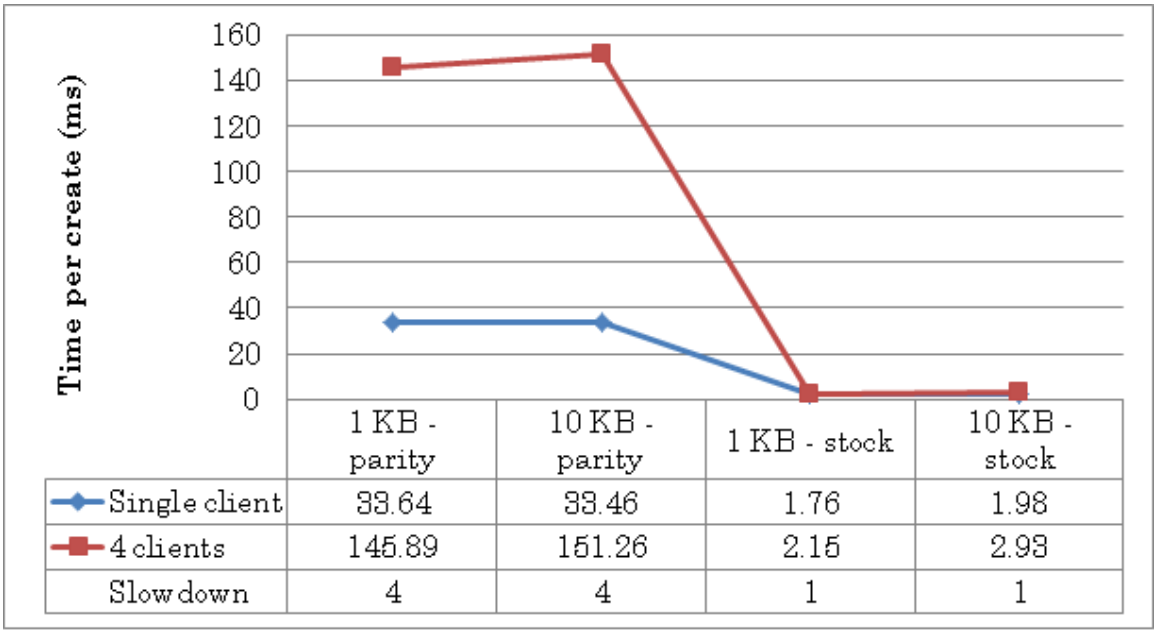
In this chapter, we demonstrated an approach to improve the reliability of distributed storage systems using parity-based data redundancy on object storage. We provided implementations for RAID4 and RAID5 configurations where the parity management is handled by the OSD emulations. In doing so, we reduced the workload on clients, reducing the performance overhead and also making the system portable and eas-



	1 KB - parity	10 KB - parity	1 KB - stock	10 KB - stock
Single client	30.73	32.37	1.71	1.87
4 clients	164.7	178.3	2.42	2.24
Slow down	5	5	1	1

FIGURE 6.3.6: Time (in milliseconds) per creating a file of various sizes and number of clients with RAID5 redundancy scheme and stock PVFS-OSD implementation (single and 4 PVFS clients, 8 OSDs)

ily configurable with other network clients and file systems without modifying the clients.



	1 KB - parity	10 KB - parity	1 KB - stock	10 KB - stock
Single client	33.64	33.46	1.76	1.98
4 clients	145.89	151.26	2.15	2.93
Slow down	4	4	1	1

FIGURE 6.3.7: Time (in milliseconds) per creating a file of various sizes and number of clients with RAID5 redundancy scheme and stock PVFS-OSD implementation (single and 4 PVFS clients, 12 OSDs)

Chapter 7

Versioning-based Unified Object Store

Storage technology has improved rapidly, particularly in terms of storage density, but storage throughput has not kept pace with advances in computational performance. This trend has led to increased demand for large-scale storage systems that aggregate and coordinate many storage devices, in turn driving the need for better abstractions to manage those storage devices. Object-based storage is a commonly used alternative of the block-based model in distributed storage systems as mentioned in Chapter 2.

Although several object-based storage models have been implemented and used as the basis for popular storage and file systems [129, 36, 40, 131], existing object-based storage models are typically designed for particular use cases or data models, making it difficult to reuse them in other contexts. This situation also makes it difficult to share a common storage pool for different big data, cloud storage or HPC storage tasks, increasing management overhead and adding complexity to the task of storage allocation for facilities with different storage needs. Ideally, each data model would

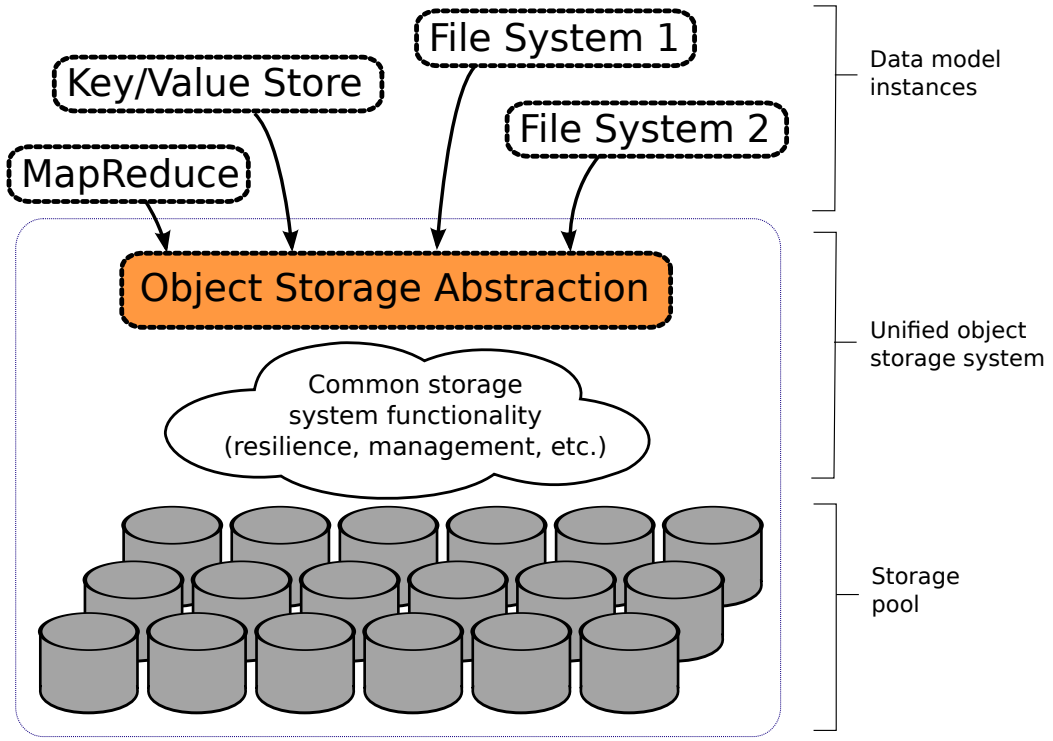


FIGURE 7.0.1: Example deployment scenario in which big data, cloud storage and HPC data models share the same storage pool via a unified object storage abstraction

coexist using a shared object storage foundation as shown in Figure 7.0.1.

To address this problem, we first identify some of the most commonly used large-scale data models in use today. The following list divides them into four categories with representative examples:

- **Parallel file systems**: Lustre [40], GPFS [115], Panasas [131], PVFS [45], Ceph [129]
- **Cloud object storage**: Amazon S3 [2], Swift [41], Rados Gateway [18, 130]
- **MapReduce**: Google File System (GFS) [66], Hadoop HDFS [120]
- **Key/value stores**: Dynamo [1], Redis [22], Hyperdex [17], Cassandra [82],

HBase [64], BigQuery [6]

Note that these data models are not necessarily mutually exclusive. For example, several parallel file systems have been extended to support MapReduce workloads. We will refer to these classifications in the remainder of this chapter for clarity. However, in order to simplify the discussion of use cases and requirements that are shared across groups of storage systems.

TABLE 7.0.1: Requirements for popular scalable storage data models

	Shared Requirements				**Distinguishing Requirements**				
	High Performance	Scalability	Fault Tolerance	Concurrent Read Access	Concurrent Write Access	Synchronization Primitives	Atomicity	Compute/Storage Locality	Record Oriented Access
Parallel File System	✓	✓	✓	✓	✓	✓	✓		
Cloud Object Storage	✓	✓	✓	✓			✓		
MapReduce	✓	✓	✓	✓				✓	
Key/Value Store	✓	✓	✓	✓	✓	✓	✓		✓

Table 7.0.1 breaks down the large-scale storage data models in terms of core requirements. The first four are general requirements that are shared across all such data models. Concurrent write access refers to the ability to have multiple processes write simultaneously to the same file, object or database. Synchronization primitives are features such as file locking [88] or conditional operations [44] that allow multiple processes to explicitly coordinate concurrent writes. Atomicity is the ability to modify data such that a write is applied in its entirety or not at all. The granularity

of atomicity can vary widely across data models. For example, cloud object storage systems may offer object-granular atomicity, key/value stores may offer per-key granularity and file systems may not offer atomicity at all except in the directory namespace. Locality allows applications to execute on server nodes with local copies of data relevant to computation. Record-oriented access is needed for storage systems that refer to units of data in terms of opaque keys rather than ranges of bytes.

In this chapter, we propose a new object-based storage API, known as the Advanced Storage Group (ASG) interface, that unifies features required to support the data models outlined above without weakening usability or limiting implementation flexibility. The contributions of this chapter are as follows:

- Identify the requirements that differentiate four key large-scale data storage models
- Propose a new object storage API that unifies the features necessary to meet those requirements
- Present a set of case studies that evaluate how the proposed API would be used as a foundation for a diverse set of storage constructs

The rest of this chapter is organized as follows. We present related studies in Section 7.1. Section 7.2 presents example drivers for this work. Section 7.3 describes the proposed ASG API and how it can be used to implement the use cases given in Section 7.2. Section 7.4 introduces the ASG API operations. We discuss how ASG API features can meet the requirements of common data models in Section 7.5. Section 7.6 shows how our approach presents a unified solution for the use cases described in Section 7.2. Section 7.7 summarizes our findings and presents potential

avenues for future work. We should also note that, the primitives in Section 7.4 were defined by the MCS division in Argonne National Laboratory from an architectural point of view before the start of work in this chapter of the thesis. Additionally, two use cases (Section 7.6.2 and Section 7.6.3) were largely written by Phil Carns and Dries Kimpe when they worked on this project at Argonne National Laboratory.

7.1 Related Work

Considering our approach to design a unified storage framework, numerous studies in the literature have similar scope. Ursa Minor [25] is a parallel file system that supports versioning-based writes. It keeps the existing object-storage interface [124] mostly intact except for introducing *slices* (i.e. fragments of object data) and it uses timestamps to distinguish different versions of data. Datamods [128] is a framework that exploits existing large-scale storage system services to support complex data models and interfaces. Datamods avoids duplicating services already provided in distributed storage systems in middleware and improves scalability since it is not limited to a single dimension at the file level. VSAM [85] supports both fixed-sized and variable-sized records depending on the application. Forks in NTFS [112] are similar to records in the ASG storage model. They are byte streams storing file data and auxiliary information such as metadata and security settings. Conditional operations are used in Amazon SimpleDB [3], Amazon DynamoDB [1], Redis [22] and Hyperdex [17].

There are a number of studies forming the technical basis of our approach for designing a unified object storage. Transactional Object Storage Device (TOSD) [43]

shows that object-based storage is a common component of many parallel file systems and it introduces three optimizations to the object-based storage model in order to serve highly concurrent workloads better: atomicity, versioning and commutativity. Goodell et al. [68] extended the POSIX API by organizing the storage around data objects in order to map complex data structures to these data objects and have direct access between the data objects and applications. Carns et al. [44] investigated conditional update operations as an alternative to distributed pessimistic locking operations in object-based storage systems.

7.2 Motivation

In this section, we discuss a number of common storage uses, that serve as the drivers for our approach.

7.2.1 Implementing POSIX Directories

In a POSIX file system, data files are located by looking them up by name in a directory. POSIX directories have the following properties. First, creating or removing a file (or subdirectory) in a directory is an atomic operation and duplicate entries are not allowed. If multiple processes try to create or remove the same entry at the same time, exactly one of them will succeed. Second, a directory entry has associated metadata, for example, the last access time or the size. In addition to create and remove, three other operations are possible on a directory: opening (lookup) of a name, updating the metadata associated with a name and renaming an entry to a new name. These operations are atomic as well. As soon as an update completes, all

processes in the system see the updated information. At any time before that, the old information is preserved. At no point will a lookup return a blend of old and new metadata. The same is true for rename. Either the old name will be in the directory or the new name, but not both.

Existing object storage models typically do not directly support directory primitives, nor do they support operations designed to implement structures and synchronization required for implementing a POSIX directory. Consequently, most data models requiring directory like indexes implement this functionality by using additional services (for example, metadata servers), using the object storage only for the actual file data.

7.2.2 Column-Oriented Key/Value Store

A column-oriented database differs from a traditional database in that records are stored in column order rather than row order, as shown in Table 7.2.1. Data for a database entry is stored in a column, each row stores the same data field of a database entry and shards (horizontal partitions of a database) represent a collection of rows. This organization improves the performance of analysis-oriented workloads in which ad hoc queries are performed over all values in a column. A column-oriented database will generate large, contiguous disk access patterns in this case because there is no need to skip over interleaved column data for each entry. In addition, each row typically has a large number of columns and not every row needs to have the same set of columns. Most column-oriented databases allow the creation of new columns at any time, simply by writing to them.

Existing object storage models generally do not support column-oriented databases,

		Column 0	Column 1	Column 2	Column 3
Shard 1	Row 0	Alice	Bob	Brad	Charles
	Row 1	Smith			Springfield
Shard 2	Row 0	111-1111		144-1144	321-4321

TABLE 7.2.1: Example organization of a column-oriented key value store

since record functionality is missing and applications are forced to manage the storage space. As an example, in T10 [124], if each attribute represents a cell of the column-oriented database, grouping certain attributes to represent the rows and columns of a column-oriented database is not an easy task, since mapping attributes to rows and columns and keeping track of mapping information is challenging.

7.2.3 HPC Application Checkpoint

HPC application workloads are characterized by bursty, highly concurrent, write-intensive I/O patterns [33]. In particular, many scientific simulations periodically write checkpoint data for application resilience. In these scenarios, all application processes typically write simultaneously to the same shared data set, as shown in Figure 7.2.1. Although the application processes are coordinated and do not generally write to overlapping byte ranges in the file, the access patterns may be highly interleaved and are not necessarily block aligned. Optimizations such as two-phase I/O [52] and I/O forwarding [27] can be used to mitigate the level of concurrency observed by the storage system, but data must still be written by many processes in order to leverage enough I/O paths to meet bandwidth requirements.

Metadata overhead and high concurrency are the key challenges for this type of concurrent write access pattern. Existing data models tried $N - N$ and $N - 1$ check-

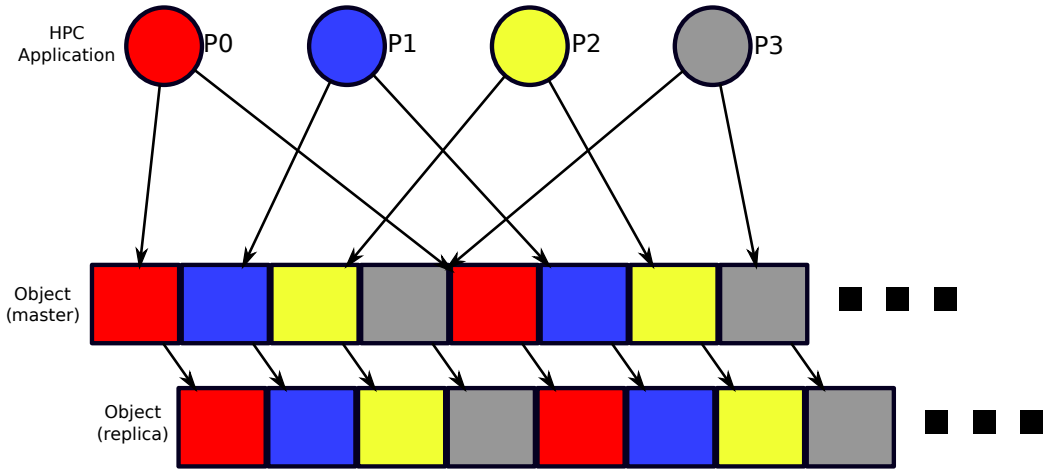


FIGURE 7.2.1: Example of an HPC application writing in parallel to a replicated object

pointing strategies [38]. $N-N$ checkpointing is the case where each process writes to a separate checkpoint file and $N-1$ checkpointing is the case where all the processes write to a shared file. Both checkpoint patterns pose challenges. $N-1$ checkpointing suffers from limited bandwidth, since all processes are trying to concurrently write to the same file. On the other hand, $N-N$ checkpointing creates a lot of files, increasing the metadata overhead.

7.3 Architecture

In this section we describe the ASG storage model, its fundamental building blocks and basic operations.

The main architecture of the ASG storage model is shown in Figure 7.3.1. The core concepts are described as follows.

- The basic building block of the ASG storage model is a *record*. Each record consists of a key, a version number, data and length of data. The key, version

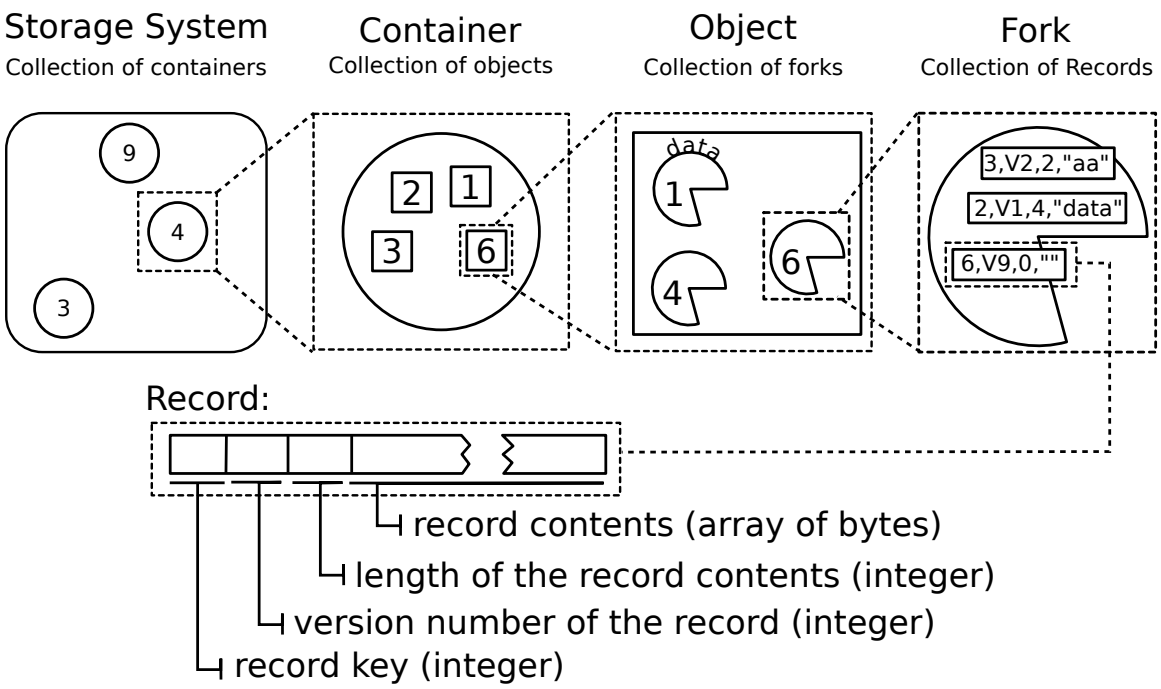


FIGURE 7.3.1: Architecture of the ASG storage model

number, and length of data are represented with integers, whereas data is an array of bytes of variable length. The key is the numerical identifier of a record. Version numbers are used to order the write operations to a record. The data field can be empty. A record at its *initial condition* will have version number *zero* and will contain no data. The records are not explicitly created in the ASG storage model, since they already exist in the system at their initial conditions before they are manipulated by the ASG storage model operations.

- A *fork* is a collection of records forming a distinct namespace for the records it contains. Each fork is identified by an integer. Forks allow related collections of data (for example, indexes, metadata or header information) to be stored alongside the primary data stream for an object with the same security, locality and atomicity [97].

- An *object* is a collection of forks. It provides a distinct namespace for the forks it contains. Each object is identified by an integer.

- A *container* is a collection of objects. It provides a distinct namespace for the objects it contains. Each container is identified by an integer. Containers partition the storage system into logical units with different security domains. As an example, each container could contain a distinct file system.

The record, fork and object identifiers in the ASG storage model are not global. For example, two different containers can have objects with the same identifiers. Similarly, two different objects or forks can have forks or records with the same identifiers. There can be a maximum of 2^{64} objects in a container, 2^{64} forks in an object and 2^{64} records in a fork in the ASG storage model, giving many options to translate the applications described in Section 7.2 to the storage model.

7.4 Operations

In this section we describe the ASG storage model operations a client can use to interact with the storage system. All of the ASG storage model operations are atomic, meaning that they either fully complete or have no effect at all, which satisfies the atomicity requirement in Table 7.0.1.

7.4.1 write

The *write* operation stores data in a sequential range of records. The input arguments to this function are location information (container, object, fork and starting record

identifiers), local buffer that stores the data to be written, number of records to be modified (range of the write operation), conditional flag and user-specified version number. The conditional flag can be set to one of the following four values in order to control the semantics of the write operation with respect to per-record version numbers:

- *NONE*: Write should succeed without checking any version number.
- *ALL*: Write should succeed only if the user-specified version number is greater than all the version numbers in the user-specified range.
- *UNTIL*: Write should *continue* until it comes across a record that has a version number greater than or equal to the user-specified version number.
- *AUTO*: In this case, the user-specified version number can be ignored. The biggest version number existing in the user-specified range is found and incremented, and the new data is written with this incremented version number.

The input data of the write operation is divided into the same number of chunks as the number of records in the user-specified range. The same length of data is written to each record in this range.

When the write operation is completed successfully, it returns the size of the written data and the newly assigned version number.

7.4.2 read

The *read* operation retrieves data from a sequential range of records. The input arguments to the read operation are location information (container, object, fork and

starting record identifiers), local buffer to store the read data, number of records to be read, conditional flag and user-specified version number. The range in the read operation is identical to the range defined in the write operation. The conditional flag of the read operation can be set to one of the following three values:

- *NONE*: Read should succeed without checking any version number or conditional flags.
- *ALL*: Read should succeed only if the user-specified version number is greater than all the version numbers existing in the user-specified range.
- *UNTIL*: Read should *continue* until it comes across a record that has a version number greater than or equal to the user-specified version number.

When the read operation is completed successfully, it returns the number of the records read in addition to the version number of these records.

7.4.3 reset

The *reset* operation returns an entity (container, object, fork or record) back to its *initial condition*. In an entity at its original condition, all the records will have version number *zero* and will contain no data. The reset operation can work on any entity of the ASG storage model. The reset operation takes in the identifier information of the entity to be reset as an input argument and it also supports conditional execution based on the existing version number and given conditional flag. The conditional flags that can be used with the reset operation are the same as the conditional flags used in the read operation. When the reset operation is completed successfully, it returns the number of entities reset.

7.4.4 probe

The *probe* operation can be used to iteratively enumerate containers within a storage system, objects within a container, forks within an object or records within a fork. It takes as inputs the identifier of the entity to probe (a container, object or fork), the identifier of the entity (a container, object, fork or record) to start probing from, local buffer to store the retrieved information and maximum number of entities for which the information will be retrieved. At the completion of the probe operation, various information about an ASG entity, such as the range of existing records in that entity, their version numbers and sizes etc., is returned.

7.5 Relation to Data Model Requirements

In this section, we show how the features provided by the ASG storage model make it possible to meet the requirements of the common data models listed in Table 7.0.1. We note that none of these features are new. The ASG storage model just presents a reusable unified API bringing these features together while minimizing complexity. The features provided by the ASG storage model and how they meet the requirements of common data models can be summarized as follows:

- *Unified byte stream and key/value storage*: The ASG storage model supports both byte-stream [131] and key/value-based storage [1]. Each byte is stored as a one-byte record. With the numerical identifiers and record contents, each record can be also used as a key/value store. As a result, the ASG storage model supports both file-based and key/value-based access models and also enables **record-oriented access** for both of these models.

- *Eliminating object attributes*: Object-based storage models, such as T10 [124], use attributes to describe the objects, meaning that object attributes are used to store metadata. In the ASG storage model, we still have metadata describing the records. However, we do not store the metadata in separate attributes as is normally done in object-based storage models. Metadata can be stored in dedicated forks, giving the opportunity to store data and metadata together, to treat metadata in the same way as data and to have simple metadata management by alleviating the need for a separate metadata API. Simplified metadata management reduces the metadata overhead and improves **scalability**.

- *Record versioning*: Versioning in the ASG storage model enables sorting writes to a record as shown in previous studies [25, 43]. As the version number changes with each write operation, highly **concurrent write operations** will be consistent and the **performance** of the system will increase.

- *Conditional operations*: The conditional read and write operation flags provide **synchronization primitives** and **atomicity** in a data model, as has been shown in a few storage models [44, 3, 1, 22, 17]. Using the conditional flags with the ASG storage model operations, multiple processes can coordinate *concurrent writes* without using any explicit locking method. Using conditional flags also ensures that each ASG operation either fully completes or has no effect, making sure the system does not end up in an inconsistent state and that **fault tolerance** is achieved.

- *Independently addressable records*: ASG is a **record-oriented** storage model similar to some other data models [16, 85]. Each entity in the ASG storage model has a numerical identifier. When ASG primitives access a record, they

explicitly use the identifiers of the enclosing container, object and fork along with the identifier of that record. As a result, each record in ASG has a distinct location and is independently accessible. Records are the smallest units of storage an operation can access in the ASG storage model. Having access to independently addressable records also makes it possible to support **concurrent read and write access** on them.

- *Fork structure*: Forks can be used to store metadata, as discussed previously. In addition, they can be used to group records that store related data together [112, 97]. This approach improves **performance** by simplifying data management and enables collecting provenance from related records in an efficient way to support **fault tolerance**.

- *Server location*: The ASG storage model exposes location information of its entities to higher-level applications. Applications either can take control of the server location of ASG entities by using this information or they can let the storage system to handle localization and choose ASG entities randomly without worrying about server locations. When the applications decide which ASG entities to use, they can move **computation closer to storage**.

7.6 Use Cases

In this section we show how ASG storage model can be used as a foundation for three example use cases.

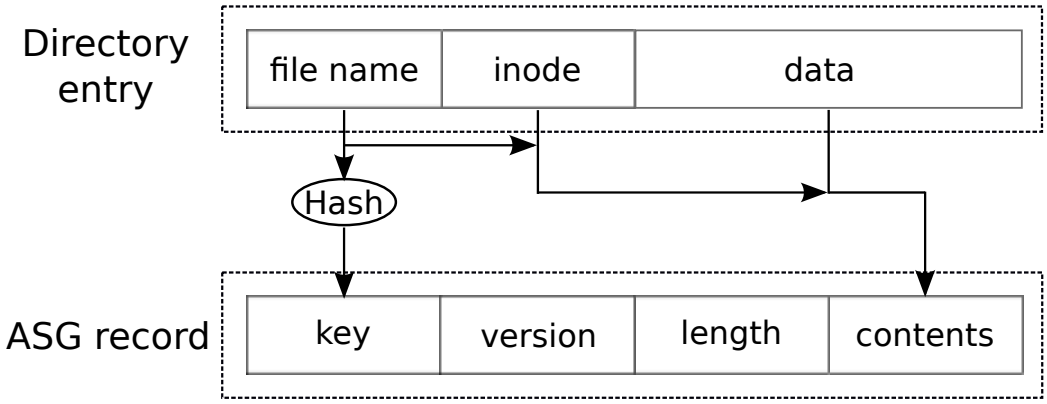


FIGURE 7.6.1: Mapping directory entries to ASG entities

7.6.1 Implementing POSIX Directories

In our first use case, we show how the ASG storage model features can be used to implement POSIX directory operations. Entries in a directory can be represented with the ASG records as shown in Figure 7.6.1. In order to map a directory entry to an ASG record, the name of the directory entry can be hashed into a record key. The name, inode information and data of the directory entry can be stored together in the ASG record. Since each ASG record is independently addressable, the uniqueness requirement of each entry in a directory can be satisfied. Indeed, one can implement POSIX directory operations using the ASG storage model. We note that other namespace and directory implementations also can be supported by using the ASG storage model, even though we show a POSIX directory implementation in this section.

In order to create a new file (or subdirectory) in a directory, an underlying ASG *write* operation is called with conditional flags. Similarly, in order to remove a file (or subdirectory) from a directory, underlying ASG *read* and *reset* operations are called with conditional flags. The ASG *read* operation returns the version number of

the ASG record representing a directory entry. Checking the version number returned by the ASG *read* operation, ASG *write* does not create a directory entry if it already has been created (nonzero version number) and ASG *reset* does not remove a directory entry if it has not been created yet (zero version number). As a result, the ASG storage model ensures the atomicity of the POSIX create and remove operations and it prevents duplicate directory entries. If multiple processes try to create or remove a directory entry at the same time, only one of them succeeds.

Other POSIX directory operations, such as updating the metadata of a directory entry and renaming an entry, can be also supported by using the ASG storage model. In order to update the metadata of a directory entry or rename a directory entry, ASG *read* and *write* operations are called with conditional flags. Again, the ASG *read* operation returns the version number of the ASG record representing a directory entry. The ASG *write* does not update the metadata of a directory entry if it has not been created yet (zero version number). While renaming a directory entry, ASG *write* creates the new directory entry with no conditional flags, meaning that it overwrites the new entry if it already exists, and ASG *reset* removes the old entry as soon as the new entry is successfully created. As a result, the ASG storage model ensures the atomicity of the POSIX update and rename operations. All processes in the system see the updated metadata information as soon as update is done and they see the old metadata information at any time before update completes. No process sees old and new metadata at the same time. For the rename operation, either the old or the new directory entry exists, not both.

In order to lookup a directory entry or to stat a directory, ASG *read* and *probe* operations are called. Similar to the previous POSIX operation implementations, the ASG *read* operation returns the version number of the ASG record representing a

directory entry. This version number is not important for the lookup operation, which returns any data available in the entry at time it was called. For the stat operation, however, ASG *probe* keeps track of this version number. Hence, if the directory is modified while the stat operation is not done yet, ASG *probe* identifies modified entries and returns updated information about them as a result of the stat operation. Using conditional operations, the ASG storage model ensures the atomicity of the lookup and stat operations by returning updated information with them. At no point old and new information for an entry is returned together.

7.6.2 Column-Oriented Key/Value Store

Table 7.6.1 shows an example of how a column-oriented key/value database might be expressed using ASG primitives. Rows are represented as ASG records, columns are represented as ASG forks and shards are represented as ASG objects. ASG records are variable-sized and any value in the database can be referenced by a unique {object ID, fork ID, record ID} triple. Since ASG *write* operation can take zero-length data as input, rows can have empty columns in the database.

TABLE 7.6.1: Example organization of a column-oriented key value store using the ASG storage model

		Column:fork 0	Column:fork 1	Column:fork 2	Column:fork 3
Shard:object 1	Row:record 0 Row:record 1	Alice Smith	Bob	Brad	Charles Springfield
Shard:object 2	Row:record 0	111-1111		144-1144	321-4321

Columns map well to forks in this example because the fork construct allows each column to be addressed independently while still ensuring that all records within a row are stored in the same object. An entire row can therefore be accessed (or added or removed) atomically. The ASG object storage model does not dictate on-disk

layout, but it is expected that the storage system would organize data on disk such that forks are contiguous in this case. Shards map naturally to objects because they partition the data set into discrete chunks that can be used to parallelize column-oriented queries, if objects are distributed across different servers.

Forks and variable-sized records provided by the ASG interface are critical to expressing this use case. Without these features, a column-oriented key/value storage system would be forced to maintain an additional mapping index to translate between row-column nomenclature and offset-size nomenclature. This translation layer not only would add complexity for the data model implementor, it would also prevent critical semantic information from being expressed to the storage system. An ASG-based storage system, for example, may recognize a linear column-oriented access pattern and adapt its underlying storage layout accordingly, while a traditional storage system making the same optimization would have to do so based on assumptions derived from generic byte range access patterns. The ASG *probe* function also leverages the additional structured data semantic information provided by fork and record-oriented access to enable efficient enumeration of both the rows and columns of a table.

7.6.3 HPC Application Checkpoint

Implementing HPC checkpointing strategies directly on top of the ASG storage model is straightforward because of its structure and explicit location control feature. One can implement both $N - N$ and $N - 1$ checkpointing strategies using the ASG storage model and thus overcome the limitations of these methods as explained in Section 7.2.3.

In the $N-N$ checkpointing method, each process writes to a separate checkpoint file. As explained in Section 7.5, the ASG storage model exposes location information of its entities to higher-level applications. Therefore, any application trying to implement $N-N$ checkpointing method can take advantage of this information to pick ASG entities that will store checkpoint data, in a manner to balance the metadata load across the system without dealing with any dedicated location servers. Additionally, as explained in Section 7.5, object attributes are eliminated in the ASG storage model and as a result metadata management is simplified. Having simple metadata management and explicit location control reduces the metadata overhead in the $N-N$ checkpointing method.

If the $N-1$ checkpointing method is implemented, each process writes to a shared checkpoint file. As explained in Section 7.5, the ASG storage model has record versioning and conditional operation features. Therefore, processes trying to write to a checkpoint file concurrently can take advantage of record versioning and conditional operations to order their writes to the checkpoint file. This strategy alleviates the need to set locks on the checkpoint file, since it is possible to update the file atomically using the ASG storage model operations. As a result, having record versioning and conditional operations makes it possible to have highly concurrent writes to the checkpoint file in the $N-1$ checkpointing method.

7.7 Summary

In this chapter, we presented a new object-based storage model, ASG, introduced its architecture and primitives, and described a couple of use cases based on this

model. As the use cases clearly show, the ASG storage model is flexible and can act as a starting point for building complex storage applications. Features supported by the ASG storage model make it possible to support requirements of common data models.

Chapter 8

Conclusion and Future Work

In this thesis, we have presented our approaches to improve energy utilization, performance, reliability and usability in a distributed storage system. To reduce energy consumption of distributed storage systems, we presented three different energy-aware node allocation methods. These methods benefit from storage network incast and make use of user metadata for allocating cloud storage system resources for each user, while balancing load (i.e. used storage space, on-time) on-demand. Each method is evaluated both theoretically and simulatively. In order to improve the metadata performance of storage systems, we presented using object collections as directories and post-creating objects for two common operations in distributed storage systems - creating a large number of files in a directory and reading from a directory with large number of files. Similarly, in order to improve the performance of data operations in storage systems, we presented an approach that performs computation on existing large-scale data in an object storage system without moving data anywhere and analyzed the outcomes of our approach. This approach has been implemented and

tested with the Hadoop and Ceph storage systems.

We then presented the implementation of a parity-based redundancy approach on object storage. We evaluated this approach in RAID4 and RAID5 configurations and showed that the performance overhead at the clients due to data redundancy operations can be reduced while also making the object storage system more portable as the reliability is handled at the storage. Finally, we presented ASG, a new flexible object-based storage framework that unifies the requirements of several different data models. We have shown that ASG features can be used to build complex storage applications by providing example use cases.

8.1 Future Work

As a future work, the proposed energy conservation techniques need to be evaluated in a real storage system with more storage nodes, more users and with different workloads. The low-energy mode for a storage node was turning it off completely. We are aware that certain modern disks support various operating modes with different power requirements. Therefore, it is worthwhile to evaluate our methods with disks supporting multiple operating modes. Additionally, proposed methods can be implemented with various types of user-metadata other than the ones used in this thesis and in a system that already prevents incast.

As part of future work for improving metadata performance, communication between OSDs can be leveraged to further improve the performance of our proposed optimization [57, 26, 108, 110]. In terms of improving the data performance, future research directions include tests with more data and larger storage systems. Addi-

tionally, metadata consistency between the MapReduce framework and the underlying storage system (with respect to checksumming mechanisms in particular) is another area of future research efforts.

There are several possible future research directions for the proposed object-based redundancy techniques. These techniques can be tested with more detailed evaluations where large writes and reconstructions are enabled. Furthermore, object attributes can be used to prioritize certain objects during reconstruction and multithreading can be implemented for better performance. Finally, the ASG unified storage framework can be used as the basis for complex storage systems and there already some studies using it [50].

Bibliography

[1] *Amazon DynamoDB*, http://aws.amazon.com/dynamodb/.

[2] *Amazon Simple Storage System (Amazon S3)*, http://aws.amazon.com/s3/.

[3] *Amazon SimpleDB*, http://aws.amazon.com/simpledb/.

[4] *Amazon Web Services*, http://aws.amazon.com/.

[5] *Apache Hadoop Project, The Apache Software Foundation*, http://hadoop.apache.org/.

[6] *BigQuery*, https://developers.google.com/bigquery/.

[7] *Booth Engineering Center for Advanced Technology, University of Connecticut*, http://www.becat.uconn.edu/hpc/hornet-cluster.

[8] *EMC Atmos Storage*, http://www.emc.com/storage/atmos/atmos.htm.

[9] *EMC Centera Storage Platform*, http://www.emc.com/data-protection/centera.htm.

[10] *EMC Isilon OneFS: A Technical Overview*, http://www.emc.com/collateral/hardware/white-papers/h10719-isilon-onefs-technical-overview-wp.pdf.

[11] *Fit all valid parametric probability distributions to data*, http://www.mathworks.com/matlabcentral/fileexchange/34943-fit-all-valid-parametric-probability-distributions-to-data/content/allfitdist.m.

[12] *Google Cloud Platform*, https://cloud.google.com/compute/.

[13] *Google Cloud Storage*, https://developers.google.com/storage/.

[14] *Hadoop Cluster Setup*, http://hadoop.apache.org/docs/r1.2.1/cluster_setup.html).

[15] *Hadoop JIRA HDFS-941*, https://issues.apache.org/jira/browse/HDFS-941).

[16] *HP OpenVMS Systems Documentation, Files-11 On Disk Structure Concepts*, http://h71000.www7.hp.com/doc/731final/4506/4506pro_001.html.

[17] *Hyperdex: A Searchable Distributed Key-Value Store*, http://hyperdex.org/.

[18] *Librados API documentation*, http://ceph.com/docs/master/api/librados/.

[19] *Microsoft Azure Cloud Computing Platform*, http://azure.microsoft.com/.

[20] *OpenStack Object Storage (Swift)*, http://docs.openstack.org/developer/swift/.

[21] *Orange File System*, http://orangefs.org/.

[22] *Redis*, http://redis.io/.

[23] *The Grid Workloads Archive, Delft University of Technology*, http://gwa.ewi.tudelft.nl/.

[24] *Using Lustre with Apache Hadoop, Sun Microsystems Inc.*, http://wiki.lustre.org/images/1/1b/Hadoop_wp_v0.4.2.pdf.

[25] Michael Abd-El-Malek, William V. Courtright, II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie, *Ursa minor: Versatile cluster-based storage*, Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4 (Berkeley, CA, USA), FAST'05, USENIX Association, 2005, pp. 5–5.

[26] Anurag Acharya, Mustafa Uysal, and Joel Saltz, *Active Disks: Programming Model, Algorithms and Evaluation*, SIGPLAN Not. **33** (1998), no. 11, 81–91.

[27] 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*, Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on, Aug 2009, pp. 1–10.

[28] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan, *An osd-based approach to managing directory operations in parallel file systems*, Cluster Computing, 2008 IEEE International Conference on, Sept 2008, pp. 175–184.

[29] ———, *Revisiting the metadata architecture of parallel file systems*, Petascale Data Storage Workshop, 2008. PDSW '08. 3rd, Nov 2008, pp. 1–9.

[30] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R. Ganger, Michael A. Kozuch, and Karsten Schwan, *Robust and flexible power-proportional storage*, Proceedings of the 1st ACM Symposium on Cloud Computing (New York, NY, USA), SoCC '10, ACM, 2010, pp. 217–228.

[31] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris, *Scarlett: Coping with skewed content popularity in mapreduce clusters*, Proceedings of the Sixth Conference on Computer Systems (New York, NY, USA), EuroSys '11, ACM, 2011, pp. 287–300.

[32] Rajagopal Ananthanarayanan, Karan Gupta, Prashant Pandey, Himabindu Pucha, Prasenjit Sarkar, Mansi Shah, and Renu Tewari, *Cloud analytics: Do we really need to reinvent the storage stack?*, Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (Berkeley, CA, USA), HotCloud'09, USENIX Association, 2009.

[33] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny, *Workload analysis of a large-scale key-value store*, SIGMETRICS Perform. Eval. Rev. **40** (2012), no. 1, 53–64.

[34] Ana Avilés-González, Juan Piernas, and Pilar González-Férez, *Scalable metadata management through osd+ devices*, Int. J. Parallel Program. **42** (2014), no. 1, 4–29.

[35] L.A. Barroso and U. Holzle, *The case for energy-proportional computing*, Computer **40** (2007), no. 12, 33–37.

[36] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel, *Finding a needle in haystack: Facebook's photo storage*, Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA), OSDI'10, USENIX Association, 2010, pp. 1–8.

[37] Anton Beloglazov and Rajkumar Buyya, *Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints*, IEEE Trans. Parallel Distrib. Syst. **24** (2013), no. 7, 1366–1379.

[38] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate, *Plfs: A checkpoint filesystem for parallel applications*, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (New York, NY, USA), SC '09, ACM, 2009, pp. 21:1–21:12.

[39] Ricardo Bianchini and Ram Rajamony, *Power and energy management for server systems*, Computer **37** (2004), no. 11, 68–74.

[40] Peter J Braam and Rumi Zahir, *Lustre: A scalable, high performance file system*, Cluster File Systems, Inc (2002).

[41] L.-F. Cabrera and D.D.E. Long, *Swift: a storage architecture for large objects*, Mass Storage Systems, 1991. Digest of Papers., Eleventh IEEE Symposium on, Oct 1991, pp. 123–128.

[42] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, *Small-file access in parallel file systems*, Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, May 2009, pp. 1–11.

[43] P. Carns, R. Ross, and S. Lang, *Object storage semantics for replicated concurrent-writer file systems*, Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on, Sept 2010, pp. 1–10.

[44] Philip Carns, Kevin Harms, Dries Kimpe, Robert Ross, Justin Wozniak, Lee Ward, Matthew Curry, Ruth Klundt, Geoff Danielson, Cengiz Karakoyunlu, John Chandy, Bradley Settlemeyer, and William Gropp, *A case for optimistic coordination in hpc storage systems*, Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (Washington, DC, USA), SCC '12, IEEE Computer Society, 2012, pp. 48–53.

[45] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur, *Pvfs: A parallel file system for linux clusters*, Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4 (Berkeley, CA, USA), ALS'00, USENIX Association, 2000, pp. 28–28.

[46] Chien-An Chen, Myounggyu Won, R. Stoleru, and G.G. Xie, *Energy-efficient fault-tolerant data storage and processing in mobile cloud*, Cloud Computing, IEEE Transactions on **3** (2015), no. 1, 28–41.

[47] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt, *Cast: Tiering storage for data analytics in the cloud*, Proceedings of the 24th International

Symposium on High-Performance Parallel and Distributed Computing (New York, NY, USA), HPDC '15, ACM, 2015, pp. 45–56.

[48] Dennis Colarelli and Dirk Grunwald, *Massive arrays of idle disks for storage archives*, Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (Los Alamitos, CA, USA), SC '02, IEEE Computer Society Press, 2002, pp. 1–11.

[49] M. Collotta and G. Pau, *Bluetooth for internet of things: A fuzzy approach to improve power management in smart homes*, Computers & Electrical Engineering **44** (2015), 137 – 152.

[50] Ward Lee H. Curry, Matthew L. and Danielson Geoff, *Motivation and design of the sirocco storage system, version 1.0*, (2015).

[51] Mark De Berg, *Lecture Notes on Advanced Algorithms*, http://www.win.tue.nl/~mdberg/Onderwijs/AdvAlg_Material/CourseNotes/lecture5.pdf, 2008.

[52] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary, *Improved parallel i/o via a two-phase run-time access strategy*, SIGARCH Comput. Archit. News **21** (1993), no. 5, 31–38.

[53] A. Devulapalli, D. Dalessandro, and P. Wyckoff, *Data structure consistency using atomic operations in storage devices*, Storage Network Architecture and Parallel I/Os, 2008. SNAPI '08. Fifth IEEE International Workshop on, Sept 2008, pp. 65–73.

[54] A. Devulapalli, D. Dalessandro, P. Wyckoff, and N. Ali, *Attribute storage design for object-based storage devices*, Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on, Sept 2007, pp. 263–268.

[55] A. Devulapalli and P.W. Ohio, *File creation strategies in a distributed metadata file system*, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, March 2007, pp. 1–10.

[56] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, Nawab Ali, and P. Sadayappan, *Integrating parallel file systems with object-based storage devices*, Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (New York, NY, USA), SC '07, ACM, 2007, pp. 27:1–27:10.

[57] Ananth Devulapalli, Iyappa T. Murugandi, Da Xu, and Pete Wyckoff, *Design of an intelligent object-based storage device*, Tech. report, Ohio Supercomputer Center.

[58] Truong Vinh Truong Duy, Y. Sato, and Y. Inoguchi, *Performance evaluation of a green scheduling algorithm for energy savings in cloud computing*, Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, April 2010, pp. 1–8.

[59] E. Felix, *Environmental Molecular Sciences Laboratory File System Statistics*, http://www.pdsi-scidac.org/fsstats/index.html, 2007.

[60] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson, *Cohadoop: Flexible data placement and its exploitation in hadoop*, Proc. VLDB Endow. **4** (2011), no. 9, 575–585.

[61] Larry Freeman, *Reducing Data Center Power Consumption Through Efficient Storage*, Tech. report, NetApp, Inc., 2009.

[62] Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan, and Ken Birman, *Optimizing power consumption in large scale storage systems*, Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (Berkeley, CA, USA), HOTOS'07, USENIX Association, 2007, pp. 9:1–9:6.

[63] John Gantz and David Reinsel, *The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east*, IDC iView: IDC Analyze the Future 2007 (2012), 1–16.

[64] Lars George, *HBase: The definitive guide*, 2011.

[65] Gerry Houlder, Jay Elrod and Mike Miller, *XOR Commands on SCSI disk drives*, X3T10/94-111r8.

[66] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The google file system*, Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (New York, NY, USA), SOSP '03, ACM, 2003, pp. 29–43.

[67] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka, *A cost-effective, high-bandwidth storage architecture*, SIGPLAN Not. **33** (1998), no. 11, 92–103.

[68] David Goodell, Seong Jo Kim, Robert Latham, Mahmut Kandemir, and Robert Ross, *An evolutionary path to object storage access*, Proceedings of the 2012 SC

Companion: High Performance Computing, Networking Storage and Analysis (Washington, DC, USA), SCC '12, IEEE Computer Society, 2012, pp. 36–41.

[69] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke, *Drpm: Dynamic speed control for power management in server class disks*, SIGARCH Comput. Archit. News **31** (2003), no. 2, 169–181.

[70] D. Harnik, D. Naor, and I. Segall, *Low power mode in cloud storage systems*, Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, May 2009, pp. 1–8.

[71] John H. Hartman and John K. Ousterhout, *The zebra striped network file system*, ACM Trans. Comput. Syst. **13** (1995), no. 3, 274–310.

[72] S. Ibrahim, Hai Jin, Lu Lu, Bingsheng He, G. Antoniu, and Song Wu, *Maestro: Replica-aware map scheduling for mapreduce*, Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on, May 2012, pp. 435–442.

[73] C. Karakoyunlu and J.A. Chandy, *Techniques for an energy aware parallel file system*, Green Computing Conference (IGCC), 2012 International, June 2012, pp. 1–5.

[74] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web*, Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '97, ACM, 1997, pp. 654–663.

[75] A. Kathpal and G.A.N. Yasa, *Nakshatra: Towards running batch analytics on an archive*, Modelling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on, Sept 2014, pp. 479–482.

[76] Rini T. Kaushik and Milind Bhandarkar, *Greenhdfs: Towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster*, Proceedings of the 2010 International Conference on Power Aware Computing and Systems (Berkeley, CA, USA), HotPower'10, USENIX Association, 2010, pp. 1–9.

[77] Rini T. Kaushik, Ludmila Cherkasova, Roy Campbell, and Klara Nahrstedt, *Lightning: Self-adaptive, energy-conserving, multi-zoned, commodity green cloud storage system*, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (New York, NY, USA), HPDC '10, ACM, 2010, pp. 332–335.

[78] Jinoh Kim and Doron Rotem, *Energy proportionality for disk storage using replication*, Proceedings of the 14th International Conference on Extending Database Technology (New York, NY, USA), EDBT/ICDT '11, ACM, 2011, pp. 81–92.

[79] Kyong Hoon Kim, Anton Beloglazov, and Rajkumar Buyya, *Power-aware provisioning of cloud resources for real-time services*, Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science (New York, NY, USA), MGC '09, ACM, 2009, pp. 1:1–1:6.

[80] Jonathan Koomey, *Growth in data center electricity use 2005 to 2010*, (2011).

[81] Elie Krevat, Vijay Vasudevan, Amar Phanishayee, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan, *On application-level approaches to avoiding tcp throughput collapse in cluster-based storage systems*, Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07 (New York, NY, USA), PDSW '07, ACM, 2007, pp. 1–4.

[82] Avinash Lakshman and Prashant Malik, *Cassandra: A decentralized structured storage system*, SIGOPS Oper. Syst. Rev. **44** (2010), no. 2, 35–40.

[83] Jacob Leverich and Christos Kozyrakis, *On the energy (in)efficiency of hadoop clusters*, SIGOPS Oper. Syst. Rev. **44** (2010), no. 1, 61–65.

[84] Johann Lombardi and Liang Zhen, *DAOS Changes to Lustre*, Presented by Intel High Performance Data Division, April 2013.

[85] Dave Lovelace, Rama Ayyar, Alvaro Sala, and Valeria Sokal, *Vsam Demystified*, first ed., IBM Corp., Riverton, NJ, USA, 2003.

[86] C Maltzahn, E Molina-Estolano, A Khurana, AJ Nelson, SA Brandt, and S Weil, *Ceph as a scalable alternative to the hadoop distributed file system*, August 2010.

[87] Paul Massiglia, *Raidbook, 6th edition: A storage system technology handbook*, Peer to Peer Communications, 1999.

[88] D.A. Menasce and Richard R. Muntz, *Locking and deadlock detection in distributed data bases*, Software Engineering, IEEE Transactions on **SE-5** (1979), no. 3, 195–202.

[89] Mike Mesnier, Gregory R. Ganger, and Erik Riedel, *Object-based storage*, IEEE Communications Magazine **41** (2003), no. 8.

[90] Madalin Mihailescu, Gokul Soundararajan, and Cristiana Amza, *Mixapart: Decoupled analytics for shared storage systems*, Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (Berkeley, CA, USA), HotStorage'12, USENIX Association, 2012, pp. 2–2.

[91] Ethan L. Miller and Randy H. Katz, *An analysis of file migration in a Unix supercomputing environment*, USENIX—Winter 1993, January 1993.

[92] X. Mountrouidou, A. Riska, and E. Smirni, *Saving power without compromising disk drive reliability*, Green Computing Conference and Workshops (IGCC), 2011 International, July 2011, pp. 1–6.

[93] Xenia Mountrouidou, Alma Riska, and Evgenia Smirni, *Adaptive workload shaping for power savings on disk drives*, Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering (New York, NY, USA), ICPE '11, ACM, 2011, pp. 109–120.

[94] David Nagle, Denis Serenyi, and Abbie Matthews, *The panasas activescale storage cluster: Delivering scalable high bandwidth storage*, Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (Washington, DC, USA), SC '04, IEEE Computer Society, 2004, pp. 53–.

[95] S. Narayan and J.A. Chandy, *Parity redundancy in a clustered storage system*, Storage Network Architecture and Parallel I/Os, 2007. SNAPI.International Workshop on, Sept 2007, pp. 17–24.

[96] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron, *Write off-loading: Practical power management for enterprise storage*, Trans. Storage **4** (2008), no. 3, 10:1–10:23.

[97] Nils Nieuwejaar and David Kotz, *The galley parallel file system*, Tech. report, Hanover, NH, USA, 1996.

[98] A.-C. Orgerie, L. Lefevre, and J.-P. Gelas, *Save watts in your grid: Green strategies for energy-aware framework in large scale distributed systems*, Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on, Dec 2008, pp. 171–178.

[99] B. Palanisamy, A. Singh, N. Mandagere, G. Alatorre, and Ling Liu, *Mapreduce analysis for cloud-archived data*, Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, May 2014, pp. 51–60.

[100] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte, *Giga+: Scalable directories for shared file systems*, Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07 (New York, NY, USA), PDSW '07, ACM, 2007, pp. 26–29.

[101] David A. Patterson, Garth Gibson, and Randy H. Katz, *A case for redundant arrays of inexpensive disks (raid)*, Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '88, ACM, 1988, pp. 109–116.

[102] Petascale Data Storage Institute, *NERSC File System Statistics*, http://pdsi.nersc.gov/filesystem.htm, 2007.

[103] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan, *Measurement and analysis of tcp throughput collapse in cluster-based storage systems*, Proceedings of the 6th USENIX Conference on File and Storage Technologies (Berkeley, CA, USA), FAST'08, USENIX Association, 2008, pp. 12:1–12:14.

[104] Eduardo Pinheiro and Ricardo Bianchini, *Energy conservation techniques for disk array-based servers*, Proceedings of the 18th Annual International Conference on Supercomputing (New York, NY, USA), ICS '04, ACM, 2004, pp. 68–78.

[105] Eduardo Pinheiro, Ricardo Bianchini, and Cezary Dubnicki, *Exploiting redundancy to conserve energy in storage systems*, SIGMETRICS Perform. Eval. Rev. **34** (2006), no. 1, 15–26.

[106] George Porter, *Decoupling storage and computation in hadoop with superdatanodes*, SIGOPS Oper. Syst. Rev. **44** (2010), no. 2, 41–46.

[107] Philipp Reisner and Lars Ellenberg, *Replicated storage with shared disk semantics*, Proceedings of the 12th International Linux System Technology Conference, sept. 2005.

[108] Erik Riedel, *Active disks - remote execution for network-attached storage*, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, November 1999.

[109] Alma Riska, Ningfang Mi, Evgenia Smirni, and Giuliano Casale, *Feasibility regions: Exploiting tradeoffs between power and performance in disk drives*, SIGMETRICS Perform. Eval. Rev. **37** (2010), no. 3, 43–48.

[110] M.T. Runde, W.G. Stevens, P.A. Wortman, and J.A. Chandy, *An active storage framework for object storage devices*, Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on, April 2012, pp. 1–12.

[111] Rupprecht, Lukas and Zhang, Rui and Hildebrand, Dean, *Big Data Analytics on Object Stores: A Performance Study*, Proceedings of 2014 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, 2014.

[112] Richard Russon and Yuval Fledel, *NTFS Documentation*, 2004.

[113] Nathan Rutman, *Map/Reduce on Lustre, Hadoop Performance in HPC Environments*, Technical report, Xyratex Technology Limited, Havant, Hampshire, UK, 2011.

[114] Ahmed Sallam, Kenli Li, Aijia Ouyang, and Zhiyong Li, *Proactive workload management in dynamic virtualized environments*, J. Comput. Syst. Sci. **80** (2014), no. 8, 1504–1517.

[115] Frank Schmuck and Roger Haskin, *Gpfs: A shared-disk file system for large computing clusters*, Proceedings of the 1st USENIX Conference on File and Storage Technologies (Berkeley, CA, USA), FAST '02, USENIX Association, 2002.

[116] Seagate Technology, LLC, *Kinetic open storage documentation*, 2014.

[117] M. Sevilla, I. Nassi, K. Ioannidou, S. Brandt, and C. Maltzahn, *Supmr: Circumventing disk and memory bandwidth bottlenecks for scale-up mapreduce*, Parallel

Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, May 2014, pp. 1505–1514.

[118] Fareha Sheikh, Syeda Umme Habiba Fazal, Fatima Taqvi, and Jawwad Ahmed Shamsi, *Power-aware Server Selection in Nano Data Center*, Proceedings of the 40th Annual IEEE Conference on Local Computer Networks, 2015.

[119] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes, *Cloudscale: Elastic resource scaling for multi-tenant cloud systems*, Proceedings of the 2Nd ACM Symposium on Cloud Computing (New York, NY, USA), SOCC '11, ACM, 2011, pp. 5:1–5:14.

[120] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, *The hadoop distributed file system*, Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (Washington, DC, USA), MSST '10, IEEE Computer Society, 2010, pp. 1–10.

[121] Konstantin V. Shvachko, *HDFS Scalability: The Limits to Growth*, vol. 35, April 2010, pp. 6–16.

[122] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao, *Energy aware consolidation for cloud computing*, Proceedings of the 2008 Conference on Power Aware Computing and Systems (Berkeley, CA, USA), HotPower'08, USENIX Association, 2008, pp. 10–10.

[123] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti, *Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage*, Proceedings of the 6th USENIX Conference on File and Storage Tech-

nologies (Berkeley, CA, USA), FAST'08, USENIX Association, 2008, pp. 1:1–1:16.

[124] T10 Technical Committee of the InterNational Committee on Information Technology Standards, *Object-Based Storage Devices - 3 (OSD-3)*, http://www.t10.org/.

[125] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J. Lang, Garth Gibson, and Robert B. Ross, *On the duality of data-intensive file system design: Reconciling hdfs and pvfs*, Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA), SC '11, ACM, 2011, pp. 67:1–67:12.

[126] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah, *Analyzing the energy efficiency of a database server*, Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '10, ACM, 2010, pp. 231–242.

[127] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami, *Srcmap: Energy proportional storage using dynamic consolidation*, Proceedings of the 8th USENIX Conference on File and Storage Technologies (Berkeley, CA, USA), FAST'10, USENIX Association, 2010, pp. 20–20.

[128] Noah Watkins, Carlos Maltzahn, Scott Brandt, and Adam Manzanares, *Datamods: Programmable file system services*, Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (Washington, DC, USA), SCC '12, IEEE Computer Society, 2012, pp. 42–47.

[129] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn, *Ceph: A scalable, high-performance distributed file system*, Proceedings of the 7th Symposium on Operating Systems Design and Implementation (Berkeley, CA, USA), OSDI '06, USENIX Association, 2006, pp. 307–320.

[130] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn, *Rados: A scalable, reliable storage service for petabyte-scale storage clusters*, Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07 (New York, NY, USA), PDSW '07, ACM, 2007, pp. 35–44.

[131] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou, *Scalable performance of the panasas parallel file system*, Proceedings of the 6th USENIX Conference on File and Storage Technologies (Berkeley, CA, USA), FAST'08, USENIX Association, 2008, pp. 2:1–2:17.

[132] A. Wildani and E.L. Miller, *Semantic data placement for power management in archival storage*, Petascale Data Storage Workshop (PDSW), 2010 5th, Nov 2010, pp. 1–5.

[133] Avani Wildani, Ethan L. Miller, and Lee Ward, *Efficiently identifying working sets in block i/o streams*, Proceedings of the 4th Annual International Conference on Systems and Storage (New York, NY, USA), SYSTOR '11, ACM, 2011, pp. 5:1–5:12.

[134] John Wilkes, *More Google cluster data*, Google research blog, November 2011, Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[135] E.H. Wilson, M.T. Kandemir, and G. Gibson, *Will they blend?: Exploring big data computation atop traditional hpc nas storage*, Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on, June 2014, pp. 524–534.

[136] C. Xu, R. Goldstone, Z. Liu, H. Chen, B. Neitzel, and W. Yu, *Exploiting analytics shipping with virtualized mapreduce on hpc backend storage servers*, Parallel and Distributed Systems, IEEE Transactions on **27** (2016), no. 1, 185–196.

[137] Shuangyang Yang, Walter B. Ligon Iii, and Elaine C. Quarles, *Scalable distributed directory implementation on orange file system*, 2011.

[138] Weikuan Yu and Omkar Kulkarni, *Progress Report on Efficient Integration of Lustre and Hadoop/YARN*, (2014).

[139] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes, *Hibernator: Helping disk arrays sleep through the winter*, Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (New York, NY, USA), SOSP '05, ACM, 2005, pp. 177–190.

[140] Qingbo Zhu, Francis M. David, Christo F. Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao, *Reducing energy consumption of disk storage using power-aware cache management*, Proceedings of the 10th International Symposium on High Performance Computer Architecture (Washington, DC, USA), HPCA '04, IEEE Computer Society, 2004, pp. 118–.

Vita

Cengiz Karakoyunlu earned his M.S. in Electrical and Computer Engineering from the University of Connecticut (Storrs, CT) in 2013 and his B.S. in Electrical and Computer Engineering from Worcester Polytechnic Institute (Worcester, MA) in 2010. He is currently working as a software engineer for Panasas, Inc. in Pittsburgh, PA. His research interests include distributed storage systems, high-performance computing and parallel file systems.