An Application-Attuned Framework for Optimizing HPC Storage Systems

Arnab Kumar Paul

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

July 28, 2020
Blacksburg, Virginia

Keywords: Parallel File Systems, Object-Based Storage, Data Management, Load Balancing, File System Indexing, Metadata Management, High Performance Computing

An Application-Attuned Framework for Optimizing HPC Storage Systems

Arnab Kumar Paul

(ABSTRACT)

High performance computing (HPC) is routinely employed in diverse domains such as life sciences, and Geology, to simulate and understand the behavior of complex phenomena. Big data driven scientific simulations are resource intensive and require both computing and I/O capabilities at scale. There is a crucial need for revisiting the HPC I/O subsystem to better optimize for and manage the increased pressure on the underlying storage systems from big data processing. Extant HPC storage systems are designed and tuned for a specific set of applications targeting a range of workload characteristics, but they lack the flexibility in adapting to the ever-changing application behaviors. The complex nature of modern HPC storage systems along with the ever-changing application behaviors present unique opportunities and engineering challenges.

In this dissertation, we design and develop a framework for optimizing HPC storage systems by making them application-attuned. We select three different kinds of HPC storage systems – in-memory data analytics frameworks, parallel file systems and object storage. We first analyze the HPC application I/O behavior by studying real-world I/O traces. Next we optimize parallelism for applications running in-memory, then we design data management techniques for HPC storage systems, and finally focus on low-level I/O load balance for improving the efficiency of modern HPC storage systems.

In the first part of this dissertation (Chapter 3), we focus on collecting and studying file system statistics from two Livermore Computing systems with 15 PiB Lustre file systems at Lawrence Livermore National Laboratory. We add to the state-of-the art in I/O understanding by providing insight into how general HPC workloads affect the performance of large-scale HPC storage systems. In the second part of this dissertation (Chapter 4), we enable dynamic partitioning approach to improve task parallelism for in-memory data analytics frameworks. The third part of this dissertation (Chapter 5 and Chapter 6) proposes scalable and novel techniques for data management in HPC storage systems. We design and develop a generalized and scalable storage system monitor `FSMonitor` (Chapter 5) for capturing and reporting events on heterogeneous large-scale storage systems. Furthermore, we propose a metadata indexing and search tool Brindexer (Chapter 6) specifically designed for large-scale HPC storage systems. Brindexer is mainly designed for system administrators to help them manage the file system effectively. In the fourth part of this dissertation (Chapter 7), we use all the findings and tools from the previous parts to develop an "end-to-end control plane" that leverages real time load information for distributed storage server global data placement while our design model leverages trace-based optimization techniques to minimize I/O load request latency between clients and servers.

An Application-Attuned Framework for Optimizing HPC Storage Systems

Arnab Kumar Paul

(GENERAL AUDIENCE ABSTRACT)

Clusters of multiple computers connected through internet are often deployed in industry and laboratories for large scale data processing or computation that cannot be handled by standalone computers. In such a cluster, resources such as CPU, memory, disks are integrated to work together. With the increase in popularity of applications that read and write a tremendous amount of data, we need a large number of disks that can interact effectively in such clusters. This forms the part of high performance computing (HPC) storage systems. Such HPC storage systems are used by a diverse set of applications coming from organizations from a vast range of domains from earth sciences, financial services, telecommunication to life sciences. Therefore, the HPC storage system should be efficient to perform well for the different read and write (I/O) requirements from all the different sets of applications. But current HPC storage systems do not cater to the varied I/O requirements. To this end, this dissertation designs and develops a framework for HPC storage systems that is application-attuned and thus provides much improved performance than other state-of-the-art HPC storage systems without such optimizations.

Dedicated to my family - Ma, Baba, Mann, Didi, Tun da, Jhilik, and Aarush; and to my friends who have become a part of my life, for their support, encouragement, motivation, and love.

Acknowledgments

This dissertation would not have been possible without the support of many people who have become an integral part of my life. I would like to take this time to express my gratitude to these awesome individuals.

My journey in the path of Computer Science started when I was in the eighth standard when my first computer came to our house. Didi was the one who first inspired me to write computer codes to solve complex problems. I also owe my interest in Computer Science hugely to Geeta ma'am and Rama Rao Sir, who were the ones to instill in me the confidence of coding and digital logic right throughout my high school days.

During my B.Tech. days at Netaji Subhash Engineering College, I am thankful to the entire CSE faculty, especially, Anupam Sir and Pritwish Sir, who taught me the fundamentals of Operating System and Computer Hardware, which form the base of my Ph.D. dissertation. My life's journey would not have turned towards higher studies without the support of my roommates and friends during B.Tech. - Soumya, Sabya Da, Debu Da, Koontal Da, Golu, Subroto Da, Pritam, Krishna, Bikram, Arijit, Prithwish, Priyankar, Arka, Arjun, Pratip, Banu, Asif, Samik, Nandu, Manashi, Lopa, Such, Nimi, Animesh, Arunava, Kathi, Raj, Arindam, Aritra, Rahul Roy, Chandrani, Moulik, and Madhu.

Next, during my M.Tech. at National Institute of Technology Rourkela, I met so many awesome individuals who shaped my life and prepared me further for my Ph.D. Firstly, special thanks to Bibhu Sir, Rath Sir, and Mohapatra Sir for letting me work on a M.Tech. topic that was outside my specialization area but would lead me to pursue Ph.D. on that topic. They were the ones who wrote recommendation letters and motivated me to pursue my Ph.D. in the U.S.. My roommates and classmates at NIT Rourkela have been the best. *Ravikant* - I miss you a lot mere bhai! Koshy, Akshay, Sandeep, Soham, Sumanta, Rajgopal, Aditi, Sai, and Hareesh, you all have given me the best moments to cherish throughout my life.

My Ph.D. journey at Virginia Tech could not have gone smoother without the support and encouragement of my advisor, Dr. Ali R. Butt. He has been a mentor, and an elder brother to me during the course of my Ph.D.. The freedom that he gave me during my

Ph.D. to pursue my research ideas, mentor students, teach classes, and brainstorm ideas for proposals has given me the much needed boost for my entire career. He has also showed me how to balance family and work, being successful at both. I am also thankful to Dr. Eli Tilevich who gave me life and Ph.D. lessons during our ping-pong sessions, sometimes even late at night in CRC. I have also been inspired by the rigorous hard work that my committee members, Dr. Dongyoon Lee and Dr. Gang Wang, put in day and night to excel in their respective research areas.

I would also like to take this opportunity to thank Dr. Wu Feng who was my first point of contact at Virginia Tech. Thanks are in order for my labmates: Kwang, Umar - my savior, Jay, Safdar and Anwar - my mentors, Bharti, Kevin, Sangeetha, Arpit, Salman, Luna, Hadeel, Nannan, Redwan, Debasmita, Jingoo, Hyogi, Arjun, Jamaal, and Michael. The interactions, collaborations, and lengthy discussions about research projects and life in general with my labmates really enriched my Ph.D. experience. DSSL rocks! Subil - In addition to being an awesome labmate, we also collaborated on research. I hope we continue our relationship even in ORNL. Breno - For the past two years we have served as the punching bag for each other's stress in the ping-pong room. I hope we can have many more collaborations in the near future.

I am extremely thankful to the wonderful staff in the Computer Science department at Virginia Tech. Special mention needs to be given to Sharon, who has helped me numerous number of times and the special bond that we have created over the years. Teresa - without you none of my conference travels would have been possible. Melanie and Robert - thank you for taking care of us whenever it was needed, and finally the tech staff, for being there always whenever I needed technical assistance for managing our lab's cluster. Thanks are also in order for the graduate school. Thank you for being there whenever I needed help with immigration documents and procedures in the graduate school.

My Ph.D. career has been majorly marked by the three summer internships that I have done. Firstly, thank you Dr. Foster, Ryan and Kyle for giving me an opportunity to work in Argonne National Laboratory in the summer of 2017. It was my first ever internship and the experience I gained from it was tremendous. The cricket sessions, meetings in Globus by the river, or the sessions in UChicago with Tyler, Yulie, Logan, Yadu, and Hong, everything was a joy. Thanks to Subhajit for providing me a home where I had wonderful roommates - Imran, Rushit, Arjun, Ojasvee, and Nehal.
Next summer in 2018, at Lawrence Livermore National Laboratory, I had a group of the most awesome mentors - Dr. Kathryn Mohror, along with Elsa and Olaf, who provided me an excellent knowledge base of large scale HPC storage systems, which form the core of my dissertation. I met a wonderful, intelligent group of people at the lab with whom I had deep technical discussions, and also ping-pong and board game challenges. Thank you Clayton for being an awesome roommate, and special thanks to Joanna and Teresa for providing me a home for the summer. I also ran my first half marathon in Livermore, wonderful memo-

ries! I would like to acknowledge Aniket, Anirudh, Dung, Eric, Erin, Goutham, Irene, Jesse, Mahta, Kaushik, Kenneth, Nawrin, Sayket, Tao, Yue, and Zach for giving me the perfect partnership to explore California.
The summer in 2019 was one of the best summer internships that I have ever had. First of all thanks to Dr. Jay Lofstead who has always been my mentor throughout my Ph.D.. He introduced me to the people at Cray during one of the dinner parties at SC 2018. The interactions I had with Charlie Carroll - the VP of storage at Cray, provided me a unique opportunity to intern at Cray in the summer of 2019. Thank you Peter, Nathan, Cory and Brian for mentoring me with the industry aspects of how HPC storage works, and to Scott, Dr. Brad Settlemeyer, and Dominic for providing me insights of storage aspects at Los Alamos National Laboratory. My summer would not have been half as enjoyable without my awesome hosts in Los Alamos - Paul and Jill, who have become a family to me. Grandma, Connor, Courtney, and Patrick have become like my extended family now. The community at Los Alamos gave me a feeling of home away from home. Thank you Ayan da, Suranjana di, Dibya da, Soumya da, Subhashish, Pranay, Trideep, Santanu da, Supratim da, and the entire cricket gang in Los Alamos. It was a wonderful time.

My family at Blacksburg! Where do I even start. The countless memories that will be more than enough to live with in this lifetime. Thank you Anamitra da and Paroma di for being patient with me during my first year in the U.S.. Nob - one of the best persuaders in the history of ...! Prasen da - thank you for coming to Blacksburg and choosing me as your roommate. I will never ever forget the life lessons that you have given me. Ranit - your witty comments have developed a better version of me over the years. Arit - you are one of the most childish (sometimes mature) and the sweetest roommate I have ever had. GB - our friendship has only grown stronger day by day. Sreeya di - thank you being an elder sister and taking care of us. Shuchi - one of the most creative persons I have met, give her any object and she will breathe life into it. Gablu da, Shreya di - I have never missed my elder sister and brother-in-law in Blacksburg because you were there. I will miss Gugli a lot. Anindo, Debbie - continue being awesome! Arnab da, Poorna di - thank you providing me the guidance whenever I needed it. Srijan da, Swarnali di - you entered late into the Blacksburg chapter but have left a profound impact on the memories that I have during my Ph.D.. Lekha - without you many memories in Blacksburg would be incomplete. Keep shining! Esha - you showed how friendship can only keep on getting stronger with every passing day. Nath - you are an awesome individual and I hope you continue to succeed further. Deba - I hope we cherish our friendship like you cherish your vehicles. Meyur - thank you for inspiring me to continue long-distance running. Subhradeep da, Debarati di - thank you for caring me like your little brother during my initial years in Blacksburg. Thanks to my hiking friends - Khushboo, Nishtha, Sandeep, Sai, Srivats, Abhinav, Shyam, and Anirudh. I also thank my cultural brigade in Blacksburg - Donu da, Bala, Megha, Yash, Sheemantini, Poorvi, and Saikat. Thanks to all the people who have become very close to my heart over the years - Sagnik da, Ritwam, Leo, Anish, Fadikar da, Shibaji da, Atashi di, Bipasha, Jamaal, Jamie, Nisaanth, Sayantan, and Wrik da. My acknowledgement cannot be

complete without being grateful to my awesome cricket buddies in Blacksburg - Padhi, Jana, Arka, Harsh, Appy, Anshul, Parag, Rahul, Swetank, Jaggu, Keshto da, Satyaki, Abhinaba da, Abhishek, Mihir, Aman, Dibyendu da, Manik, Somya, and Kartik. Special mention to the Bangladesh community in Blacksburg for lighting up a Bengali spirit that was hidden in me - Yasin, Toyon, Anamul, Mahbub, Sajal da, and Sazzad. I am also extremely thankful to AID - Blacksburg for providing me a much needed platform to exercise my societal duties.

Last but not least, the most important people that I owe my entire Ph.D. to is my family. Ma, baba I know how much you both have struggled to let me dream big and follow my ambitions. You are one of the strongest people I have met and I just pray that I can be emotionally at least half as strong in my life as you are. Mann - thank you for coming to my life. In this dissertation, the technical writing is mine but the emotional writing is yours. You have kept me sane and provided me with much more emotional support than a person can ever crave for. I promise that I will try to make up for all the years of us staying apart. I love you and thank you for always being there. Didi, Tun da - thank you for always removing whatever obstacles have come in my journey. I could be brave in my decisions knowing that you always have my back. Jhilik and Aarush - I am as proud a *Mama* can ever be. You both are the joy in my life.

Funding Acknowledgment My research was supported by National Science Foundation (CNS-1615411, CNS-1565314/1838271, CCF-0746832, CNS-1016408, CNS-1016793, OAC 1835890, CCF-1919113, CNS-1405697 and CNS-1422788), CS at Virginia Tech, Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357, research credits provided by Amazon Web Services, resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725, and auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Many thanks to all for making this dissertation possible.

Declaration of Collaboration In addition to my advisor Ali R. Butt, this dissertation has benefited from many collaborators, especially:

- Kathryn Mohror, Elsa Gonsiorowski, Olaf Faaland, and Adam Moody contributed to the I/O analysis from the traces in Lawrence Livermore National Laboratory in Chapter 3 of this dissertation, and have been part of multiple joint papers [154, 155, 157].

- Wenjie Zhuang, Luna Xu, Min Li, and M. Mustafa Rafique contributed to the optimization in in-memory data analytics in Chapter 4.

- Ryan Chard, Kyle Chard, Ian Foster, and Steven Tuecke contributed to the development of the file system monitor in Chapter 5, and have been part of multiple joint papers [152, 153, 156].

- Brian Wang, Nathan Rutman, and Cory Spitz contributed to the design of a metadata indexer in Chapter 6.
- Bharti Wadhwa, Sarah Neuwirth, Feiyi Wang, and Sarp Oral contributed to the load balancing work in Chapter 7, and have been part of multiple joint papers [151, 158, 196].
- Jon Bernard and Kirk W. Cameron have also contributed to the design of the load balancer in the paper [196] which is part of the Chapter 7.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High performance computing (HPC) storage systems are increasingly becoming important. From life sciences and financial services to manufacturing and telecommunications, organizations are finding that they need not just more storage, but high-performance storage to meet the demands of their data-intensive workloads. This has resulted in a massive amount of data generation (order of petabytes), creation of billions of files, and thousands of users acting on large-scale distributed high performance computing (HPC) storage systems. According to a recent report from National Energy Research Scientific Computing Center (NERSC) [14], over the past 10 years, the total volume of data stored at NERSC has grown at an annual rate of 30 percent. This massive rate of data generation has resulted in an increasing need for high performance distributed storage systems like Apache Spark [211, 212], Ceph [201], GlusterFS [46], IBM Spectrum Scale [168] (formally known as GPFS) and Lustre [47].

The dependency on HPC storage systems has resulted in input-output (I/O) operations becoming the bottleneck for application performance. The current trend for high performance computing (HPC) systems is that processor performance improves at a rate of 20% per year, while disk access time improves by only 10% every year [85]. As a result, massively parallel HPC applications can suffer from imbalance in computation and I/O performance, with I/O operations becoming a limiting factor in application efficiency [105]. To mitigate this problem, much effort is needed to implementing high performance parallel file systems to support the I/O needs of HPC applications.

HPC storage systems are designed to distribute file data across multiple servers so that multiple clients can access file system data in parallel. Typically, they consist of *clients* that read or write data to the file system, *data servers* where data is stored, *metadata servers* that manage the metadata and placement of data on the data servers, and networks to connect these components. Data may be distributed (divided into stripes) across multiple data servers to enable parallel reads and writes. This level of parallelism is transparent to the clients, for whom it seems as though they are accessing a local file system. This entire storage system stack (clients, metadata servers, and data servers) can be exploited to optimize the performance of distributed storage systems.

1.1 Motivation

Several factors affect the I/O performance of big data HPC applications. First, the number and kinds of applications that an HPC storage system supports is increasing rapidly [213], which leads to increased resource contention and creation of hot spots where some data or resources are consumed significantly more than others. Second, the underlying storage systems, e.g., Ceph [201], GlusterFS [46], and Lustre [47], are often distributed, and adopt a hierarchical design comprising thousands of distributed components connected over complex network topologies. Managing and extracting peak performance from such resources is nontrivial. With changing application characteristics, static approaches (e.g., [68, 198]) are no longer sufficient, necessitating dynamic solutions. Third, the storage components can develop load imbalance across the I/O servers, which in turn impacts the performance and time to solution for the big data problem. Moreover, big data science relies on sophisticated workflows that encompass a wide variety of storage locations as data flows from instruments to processing resources and archival storage. Each storage location may be completely independent of one another, managed in separate administrative domains and providing heterogeneous storage interfaces such as hierarchical POSIX stores, high-performance distributed storage, persistent tape storage, and cloud-based object stores. Data may be stored for short to long periods on each, be accessible to dynamic groups of collaborators, and be acted upon by various actors. As data generation volumes and velocities continue to increase, the rate at which files are created, modified, deleted, and acted upon (e.g., permission changes) make manual oversight and management infeasible.

There is a plethora of application-specific parameters that impact runtime performance in Apache Spark, such as tasks parallelism, data compression and executor resource configuration. In typical data processing systems, the input (or intermediate) data is divided into logical subsets, called partitions. Specifically, in Spark, a partition can not be divided between multiple compute nodes for execution, and each compute node in the cluster processes one or more partitions. Moreover, a user can configure the number of partitions and how the data should be partitioned (i.e., hash or range partitioning schemes) for each Spark job. Sub-optimal task partitioning or selecting a non-optimal partition scheme can significantly increase workload execution time. For instance, if a partition strategy launches too many tasks within a computation phase, this would lead to CPU and memory resource contention, and thus lose performance. Conversely, if too few tasks are launched, the system would have low resource utilization and would again result in reduced performance.

Even though HPC parallel file systems such as Lustre provide much higher bandwidth and reliability than other options, continual improvement of parallel file systems is needed to reduce the impact of the increasing I/O performance bottleneck. File system designers need a comprehensive understanding of the I/O workloads on current HPC systems so that they can design successful next-generation file systems. Additionally, HPC system designers and system administrators need to understand both file system and application behavior so that they can design and tune HPC systems to run as efficiently as possible. Traditionally, I/O and storage analysis efforts have concentrated on analyzing application I/O behavior from application-level statistics [88, 89, 97, 122, 139, 147, 169], and there have been significant studies on the I/O workloads of large scale systems [50, 51, 52, 105, 123, 199, 200] which identify potential I/O bottlenecks in applications and suggest improvements to HPC users. However, while these studies can give clues about how particular applications utilize and stress HPC file systems, they do not give true insight into storage system performance under a general load. Thus, in order to draw meaningful conclusions about how to improve parallel file system designs for all users of an HPC system, we need to analyze statistics captured from the file system itself (from file system data on compute nodes and from data on servers that manage the I/O requests from HPC applications) independently of user applications.

1.2 Application-Attuned HPC Storage Systems

To address the above issues, this dissertation proposes, designs, and implements an application-attuned framework for optimizing HPC storage systems. This dissertation focuses on exploiting the different layers in the storage stack of HPC storage systems, namely, the clients on which applications are run, the metadata servers and the storage servers. The different HPC storage systems which are optimized here are - Apache Spark, Lustre file system, and Ceph object store. However, the approaches mentioned in this dissertation are applicable for any hierarchical HPC storage system. A plethora of different kinds of applications are explored, ranging from I/O intensive workloads to big data analytics workloads. The overarching goal of this dissertation is to improve the efficiency and flexibility of HPC storage systems by making them application-aware, and also improve the performance of applications by providing a transparent interface for efficient interaction with the underlying storage infrastructure, and making the storage software layer fully-aware of both the workload characteristics and the underlying storage heterogeneity.

Next, we define the various optimizations performed in achieving an application-attuned framework for the various HPC storage systems.

1.2.1 Understanding HPC Application I/O Behavior Using System Level Statistics

Analyzing HPC Application I/O behavior is a very important work to be done to understand what kinds of optimizations to perform on HPC storage systems. Some of the earliest studies of HPC file systems utilizing system statistics were in 2010 [175, 225]. These works analyzed the deployment of Lustre file systems and lessons learned from them. However, to the best of our knowledge, there have not been efforts which analyze file system statistics in an application-agnostic manner to understand how to support a general HPC workload. Lawrence Livermore National Laboratory (LLNL) is home to a variety of clusters that utilize Lustre file systems as primary storage, and millions of jobs run on these systems. In this effort, we take advantage of the Lustre resources at LLNL and perform a study of general application behavior using file system statistics from Lustre.

In the first part of this dissertation, we collect and study file system statistics from two Livermore Computing systems with 15 PiB Lustre file systems at LLNL, namely `Quartz` and `Cab`[1] [154, 155, 157]. We collect two types of data from these systems. - *Aggregate Job Statistics* and *Time-Series Job Statistics*. We analyze these system statistics without knowledge of the types of applications running on these systems with the goal of answering questions like *What are the typical I/O characteristics of I/O-heavy jobs?*, *How do I/O operations of jobs affect the metadata server?*, and *How does the size of application output files correlate with compute node memory size?* Our study of application-agnostic I/O statistics from the Lustre file system contributes to the state-of-the-art in I/O behavior understanding. We provide insight into how general HPC workloads affect the performance of file systems, which can aid system architects in improving file system and storage system designs and system administrators in tuning existing systems and advising users of best practices. These improvements will help to alleviate the I/O imbalance in HPC systems and increase the overall efficiency of HPC applications.

1.2.2 Optimizing Data Partitioning for In-Memory Data Analytics Frameworks

Inferior partitioning can lead to serious data skew across tasks in Apache Spark, which would eventually result in some tasks taking significantly longer time to complete than others. As data processing frameworks usually employ a global barrier between computation phases, it is critical to have all the tasks in the same phase finish approximately at the same time, so as to avoid stragglers that can hold back otherwise fast-running tasks. The right scheme for data partitioning is the key for extracting high performance from the underlying hardware resources. However, finding a data partitioning scheme that gives the best or highest performance is non-trivial. This is because, data analytic workflows typically involve complex algorithms, e.g., machine learning and graph processing. Thus, the resulting task execution plan can become extremely complicated with increasing number of multiple computation phases. Moreover, given that each computation phase is different, the optimal number of partitions for each phase can also be different, further complicating the problem.

In the second part of this dissertation, we propose CHOPPER [163], an auto-partitioning system for Spark[2] that automatically determines the optimal number of partitions for each

[1]Cab was decommissioned in June 2018.

[2]We use Spark as our evaluation DAG-based data processing framework to implement and showcase the

computation phase during workload execution. Chopper alleviates the need for users to manually tune their workloads to mitigate data skewness and sub-optimal task parallelism. Our proposed dynamic data partitioning is challenging due to several reasons. First, since Spark does not support changing tasks parallelism parameters for each computation phase, Chopper would need to design a new interface to enable the envisioned dynamic tuning of task parallelism. Second, the optimal data partitions differ across different computation phases of workloads. Chopper needs to understand application characteristics that affect the task parallelism in order to select an appropriate partitioning strategy. Finally, to adjust the number of tasks, Chopper may introduce additional data re-partitioning, which may incur extra data shuffling overhead that has to be mitigated or amortized. Thus, a careful orchestration of the parameters is needed to ensure that Chopper's benefits outweigh the costs.

1.2.3 File System Monitoring for Large Scale Storage Systems

Many small-scale local storage systems provide mechanisms to detect and report data events, such as file creation, modification, and deletion. Tools such as inotify [124], kqueue [108], and FileSystemWatcher [131] enable developers and applications to respond to data events. However, such tools are not scalable and suitable for large-scale parallel file systems, like, Lustre [12] and IBM's Spectrum Scale [168] (formally known as GPFS), and object-based storage systems, like Ceph [201] and Gluster [79]. Such large-scale storage systems often maintain an internal metadata collection catalog, such as Lustre's *Changelog*, IBM Spectrum Scale's *mmaudit*, Ceph's *Journal*, and Gluster's *libgfchangelog*, which enables developers to query data events. However, each of these tools provides a unique API and event description language. For example, a file creation action may be recorded as a *01CREAT* event in Lustre's Changelog, and an *openc* event in Ceph's Journal. Furthermore, the difference in the architecture of parallel file systems and object-based storage systems makes developing a generalized file system event monitoring tool for large-scale storage systems non-trivial.

In the third part of this dissertation, we present a generalized, and scalable, storage system monitor, called `FSMonitor`, for File System Monitor [152, 153, 156]. `FSMonitor` provides a common API and event representation for detecting and managing data events from different large-scale storage systems (parallel file systems and object-based storage systems). We have architected `FSMonitor` with a modular Data Storage Interface (DSI) layer to plug-in and use custom monitoring tools and services. We leverage the internal metadata collection catalog found in such storage systems to develop a scalable monitor architecture that is capable of detecting, resolving, and reporting these events. We choose Lustre as our implementation platform for parallel file systems because Lustre is used by 60% of the top 500 supercomputers [72]. For our implementation of `FSMonitor` on object-based storage system, we select

effectiveness of Chopper. We note that the proposed system can be applied to other DAG-based data processing framework, such as Dryad [94].

Ceph as it is one of the most popular and open-source object stores, with companies such as Bloomberg, Cisco, and Deutsche-Telekom using Ceph as their storage backend [5]. While our scalable monitor has so far only been applied to Lustre and Ceph storage systems, its design makes it applicable to other large-scale storage systems with metadata catalogs, such as IBM's Spectrum Scale and Gluster.

1.2.4 Efficient Metadata Indexing for HPC Storage Systems

Metadata indexing on large scale HPC storage systems presents a number of challenges. *First, scaling metadata indexing technology from local file systems to HPC storage systems is very difficult.* In local file systems, the metadata index has to index only a million files, and thus can be kept in-memory. However, in HPC systems, the index is too large to reside in-memory. *Second, the metadata indexing tool should be able to gather the metadata quickly.* The typical speed for file system crawlers is in the range of 600 to 1,500 files/sec [42]. This translates to 18 to 36 hours of crawling for a 100 million file data set. A large scale HPC storage system can often contain a billion files, which implies crawl time in the order of weeks [42]. *Third, the resource requirements should be low.* Existing HPC storage system metadata indexing tools such as LazyBase [62] and Grand Unified File-Index (GUFI) [9] require dedicated CPU, memory, and disk hardware, making them expensive and difficult to integrate into the storage system. *Fourth, metadata changes must be quickly re-indexed to prevent a search from returning inaccurate results.* It is difficult to keep the metadata index consistent because collecting metadata changes is often slow [181] and therefore, search applications are often inefficient to update.

In the fourth part of the dissertation, to address these issues in HPC storage system metadata indexing, we present an efficient and scalable metadata indexing and search system, BRINDEXER [159]. BRINDEXER enables a fast and scalable indexing technique by using a *leveled partitioning approach* to the file system. Leveled partitioning is different and more effective than the *hierarchical partitioning approach* used in state-of-the-art indexing techniques discussed above. BRINDEXER uses an in-tree indexing design and thus mitigates the issue of maintaining metadata consistency outside the file system. BRINDEXER also uses RDBMS to store the index which makes querying easier and more effective. To overcome the drawback of slow re-indexing process, BRINDEXER uses a changelog-based approach to keep track of metadata changes in the file system.

1.2.5 I/O Load Balancing for Big Data HPC Applications

Load balancing for HPC storage systems is crucial and is being actively studied in recent works [71]. Extant systems typically attempt to perform load balancing by either having limited support for read shedding to redirect read requests to replicas of the primary copy, e.g., in Ceph [135], or performing data migration. Alternatively, per-application load balancing has also been considered to balance the load of an application across the various I/O servers [198]. These existing approaches lack a global view of all the components in the hierarchical structure of the system, and mainly focus on only a small subset of metrics (e.g., only the storage capacity, and not performance of the components). Thus, these approaches cannot guarantee that the aggregate I/O load of multiple big data applications concurrently executing atop a parallel file system (with bursty behavior) is evenly distributed.

In the final part of this dissertation, we address the load imbalance problem in Lustre by enabling a global view of the statistics of key components [151, 158, 196]. We select Lustre to showcase our approach as Lustre is deployed on 60 of the top 100 fastest supercomputers [72], and improving its performance will benefit a wide range of applications and users. We go beyond just network load balancing, e.g., as in NRS [166], or per-application approaches, e.g., as in access frequency-based solutions [198], to ensure that the Lustre Object Storage Targets (OSTs) that actually store and serve the data along with other I/O system components are load balanced. We leverage the existing hierarchy of Lustre to avoid introducing additional performance bottlenecks, and co-locate the global component of our load balancer on Lustre's Metadata Server (MDS) that has a global view of all other components.

1.3 Research Contributions

From the above five aspects, *we demonstrate in this dissertation that we can optimize the performance of HPC storage systems by making them application-attuned and exploiting the different layers in the storage stack of HPC storage systems.*

To optimize clients, Apache Spark is used as the storage system and for other layers, Lustre file system and Ceph object store are used. Lustre file system [142] is one of the most widely-used parallel file systems, supporting ∼60% of the top supercomputers in the latest Top-500 list (June, 2020) [72]. Ceph is one of the most popular and open-source object stores, with companies such as Bloomberg, Cisco, and Deutsche-Telekom using Ceph as their storage backend [5].

Overall, this dissertation proposes innovative, systemic and algorithmic approaches to tackle the inefficiency and inflexibility of the data management strategies in modern HPC storage system stack. In the following, we highlight the specific research contributions in this dissertation.

Analyzing system level statistics to understand HPC application I/O behavior
Improving I/O performance has become the most important factor in modern I/O bound HPC applications. Therefore, understanding the I/O behavior of HPC applications is very important for system administrators, file system developers, and HPC users. In this work, we collect Lustre file system server level statistics from two clusters, Cab and Quartz at Lawrence Livermore National Laboratory, for a period of

three years and analyze the statistics in an application-agnostic manner. Our studies indicate interesting results which show that most jobs are write-intensive, showing the importance of improving file system write performance. Our analysis also lead us to believe that focus should be on jobs which run for short duration as the majority of the jobs run for less than an hour. Also, there should be efforts to educate HPC users to develop applications which perform efficient writes. This would improve I/O performance as well as help in reducing I/O contention among jobs. We believe that our analysis will help all HPC practitioners to build better file systems and utilize it more effectively.

Improving the performance of in-memory data analytics frameworks by optimizing data partitioning Here, we design CHOPPER, a dynamic partitioning approach for in-memory data analytic platforms. CHOPPER determines the optimal number of partitions and the partitioner for each stage of a running workload with the goal of minimizing the stage execution time and shuffle traffic. CHOPPER also considers the dependencies between stages, including join and cogroup operations, to further reduce shuffle traffic. By minimizing the stage execution time and shuffle traffic, CHOPPER implicitly alleviates the task data skew using different partitioners and improves the task resource utilization through optimal number of partitions. Experimental results demonstrate that CHOPPER effectively improves overall performance by up to 35.2% for representative workloads compared to standard vanilla Spark.

Building a scalable file system monitor for large scale storage systems In this work, we present `FSMonitor`, a generic and scalable file system monitor for capturing and reporting events on heterogeneous large-scale storage systems. `FSMonitor` uses a three-layer approach to file system event monitoring. The lowest layer, *DSI*, interacts with a file system to detect events and sends them to the middle layer, *Resolution*. Here events are resolved to their absolute path names and aggregated to be sent to the upper layer, *Interface*. The Interface layer stores aggregated events which can be accessed by clients via the `FSMonitor` API.

Designing an efficient metadata indexer for HPC storage systems In this work, we present BRINDEXER, a metadata indexing tool for large-scale HPC storage systems. BRINDEXER has an in-tree design where it uses a parallel leveled partitioning approach to partition the file system namespace into disjoint sub-trees. BRINDEXER maintains an internal metadata index database which uses a 2-level database sharding technique to increase indexing and querying performance. BRINDEXER also uses a changelog-based approach to keep track of the metadata changes and re-index the file system. BRINDEXER is evaluated on a 4.8 TB Lustre storage system and is compared with state-of-the-art GUFI and Robinhood engines. BRINDEXER improves the indexing performance by 69% and the querying performance by 91% with optimal resource utilization.

Developing an efficient I/O load balancer to manage HPC applications In this work, we present the design of an "end-to-end control plane" to optimize parallel and distributed HPC I/O systems, such as Lustre, by providing efficient load balancing across storage servers. Our proposed system provides global view of the system, enables coordination between the clients and servers, and handles the performance degradation due to resource contention by considering operations on both clients as well as servers. Our implementation provides a balanced distribution of load over OSTs and OSSs in the Lustre file system, and is able to handle both PFL and non-PFL layouts for files.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2 we introduce the background technologies and state-of-the-art related work that lay the foundation of the research conducted in this dissertation. Chapter 3 presents an analysis of HPC application behavior using real-world I/O statistics. Chapter 4 presents a mechanism for optimizing data partitioning for in-memory data analytics frameworks. Chapter 5 and Chapter 6 present a scalable file system monitor and an efficient metadata indexer for large scale HPC storage systems. Chapter 7 introduces an "end-to-end framework" to balance I/O load for HPC applications. Chapter 8 concludes and discusses the future directions.

Chapter 2

Background

In this chapter, we provide background required for various aspects of our dissertation. This dissertation is focused on creating an application-attuned framework for optimizing HPC storage systems by applying different simple yet effective application-aware strategies and techniques to existing storage solutions, to bridge the gap between modern data-intensive applications with the storage systems. This chapter summarizes the state-of-the-art research that is closely related to the major theme described in the previous chapter. We also compare them against our work by emphasizing the effectiveness, novelty, and benefits of proposed workload-aware techniques and algorithms in this dissertation.

2.1 Analysis of I/O Behavior of HPC Workloads

One of the earliest work in analyzing I/O characteristics was done by Pasquale et al. [147] where they studied the production workload of San Diego Supercomputing Center's Cray YMP. I/O analysis of the I/O intensive applications revealed that I/O patterns are predictive. Nieuwejaar et al. [139] also studied file access characteristics on parallel file system in 1996. Hus et al. [89] in 2001 focused on analyzing I/O behavior from the application' perspective, whereas our work emphasizes on the system side I/O behavior in an application-agnostic manner. Hsu et al. in 2003 [88] studied the I/O traffic in personal computers and server workloads. Even on small-scale personal computers, the analysis of I/O traffic showed similar bursty patterns as our result on large-scale LLNL supercomputers.

In 2004, Wang et al. [199] analyzed workloads in a LLNL cluster. However, they focused on studying application traces to understand the types and sizes of file requests sent by application, the size of file reads and writes, node-aware locality and utilization of I/O bandwidth by nodes. The first ever Common Internet File System (CIFS) workload analysis was performed by Leung et al. [110] in 2008. Thereafter in 2009, Carns et al. did three studies [50, 51, 105], all targeting applications in very large scale storage systems at Argonne

National Laboratory (ANL). In these three works, they studied application traces from Darshan I/O characterization tool [104], how components work together to provide I/O services to applications in Intrepid system [74], and techniques to optimize small file accesses in parallel file systems for very large scale systems. 2009 was one of the earliest times when petascale I/O workloads were studied, but all of these studies targetted application traces. The first work to study failure logs in a large scale storage system was also done in 2009 by Taerat et al. [183], where they analyzed Blue Gene/L failure log data at both system and application-levels.

In 2010, Zhao et al. [225] and Shipman et al. [175] studied the Lustre file system. While, Zhao et al. [225] worked on a prediction model to accurately predict I/O bandwidth in Lustre file system, Shipman et al. [175] talked about how the world's largest Lustre file system in 2010 was deployed in the Spider system at Oak Ridge National Laboratory (ORNL). Though both of these works talked about Lustre file system, none studied the server side characteristics of large scale deployment of Lustre file system. Oral et al. [144] extended Shipman et al.'s work [175] at ORNL by discussing the lessons learned from deploying large-scale parallel file systems in Oak Ridge Leadership Computing Facility (OLCF).

Kim et al. in 2010 [97] studined I/O needs for applications on HPC clusters. This study involved running synthetic workloads and observing system behavior which is different from our analysis which studies more than 4 million HPC jobs. Carns et al. [52] in 2011, studied an application-biased I/O characterization by performing a two month study on Interpid. In 2012, Saini et al. [169] studied the I/O behavior of five NASA applications on Lustre file system.

In 2016, Luu et al. [128] and Gunasekaran, et al. [81] discussed the application level I/O behavior at production scale at ANL and ORNL respectively. They however do not go into details regarding the characteristics of the storage servers in production scale. Using application-level behavior, Liu et al. [119] and McKenna et al. [130] in 2016, and Wyatt et al. [205] in 2017 developed machine learning algorithms to coordinate I/O traffic on large scale shared storage systems. All of these works focus on managing I/O per-application basis without considering system-side statistics.

Lim et al. [114] in 2017 studied system-side statistics, but only considering metadata operations without being application-agnostic. We have taken this work one step further by studying both metadata as well as storage server statistics without being dependent on application characteristics. Lockwood et al. [123] built TOKIO in 2018 which uses analysis results to quantify the degree of I/O contention and the benefit to users to migrate to burst buffers. The same authors in 2018 provided an extensive analysis [122] from the tools used in TOKIO. But this analysis like many past studies depends on application knowledge from Darshan. IOMiner [200] was developed by Wang et al. which provided a unified interface to query I/O statistics and analyze them. Zhou et al. [227] in 2018 and Yang et al. [208] in 2019 use the application-level I/O characteristics to build an in-memory computing framework and an end-to-end I/O monitoring solution.

2.2 Optimization in Big Data Processing Frameworks

A number of recent works have focused on improving the performance of big data processing frameworks by designing better built-in data partitioning.

Shark [206] supports a column-oriented in-memory storage, using RDDs [212], to efficiently run SQL queries and iterative machine learning functions. This is achieved by re-planning query execution mid-query if needed. Spartan [91] automatically partitions the data to improve data locality in processing large multi-dimensional arrays in a distributed setting. It transforms the user code into an expression graph based on high-level operators, namely map, fold, filter, scan and join_update, to determine the communication costs of data distribution between the compute nodes.

Re-partitioning MapReduce tasks has been actively studied [67, 76, 103, 113]. PIKACHU [76] improves load balancing of MapReduce [67] workloads on clusters with heterogeneous compute capabilities. It specifically targets the reduce phase of MapReduce execution and schedules jobs on slow and fast nodes such that jobs completion times are evened out across all nodes. On the other hand, Stubby [113] optimizes the given MapReduce workflow by combining multiple MapReduce jobs into a single job. It searches through the plan space of a given workflow and applies multiple optimizations, such as vertical packing to combine map and reduce operations from multiple jobs for reducing the network traffic. Similarly, SkewTune [103] mitigates the skewness in MapReduce applications by first detecting if a node has become idle in a cluster and then by scheduling the longest job on the idle node [215, 216, 217, 218, 219, 223].

The Hardware Accelerated Range Partitioning (HARP) [203] technique leverages specialized processing elements to improve the balance between memory throughput and energy efficiency by eliminating the compute bottlenecks of the data partitioning process. HARP makes the case that using dedicated hardware for data partitioning outperforms its software counterparts and achieves higher parallelism.

Several other works also propose solutions to optimize the data partitioning problem in order to improve the processing and storage performance of multi-processor systems [61, 102, 189, 226], cloud storage systems [173, 191, 209], database systems [136, 164, 167, 204], and graph processing systems [58, 80, 174, 187].

2.3 Monitoring of Large Scale Storage Systems

Event monitoring is a critical component that enables many software projects, from software-defined cyberinfrastructure [56], to auditing programs [41]. These projects all share a common requirement for file system level events to reliably raise notifications to the tool.

Many tools have been developed to monitor file systems and detect events. A common example is `inotify` [124], a tool for unix-based systems that can attach kernel-level listeners to directories to identify common file events. Similar tools have been developed for Windows: `FileSystemWatcher` [131], `kqueue` in FreeBSD [108], `FSEvents` in macOS [37], and `fanotify` for Android [116]. These tools are typically limited in the number of directories that can be concurrently monitored and do not have a uniform definition for file system events. The `inotify` monitor places a monitor on each directory it monitors. These monitors each require 1kb of memory.

The Python Watchdog module [165] provides a common interface to invoking multiple monitors, such as `inotify`, `kqueue`, and `fsevents`, as well as a polling option that can be deployed. Although this tool does not standardize the event interface, it does provide a common API to using each tool. Facebook's *Watchman* [73], *FSWatch* [64] and *FSMon* [228] are similar to Python's watchdog module which provide a common interface. But all of these tools rely on the operating system facilities for file system event notification, and therefore cannot be used to develop a scalable monitoring solution for distributed file system.

Monitoring large-scale Lustre file systems requires the use of specialized tools [133]. An example for monitoring Lustre is the Robinhood Policy Engine [107] which is capable of collecting events from Lustre file systems and using them to drive a policy engine to automate data management tasks, such as purging old files, for example. Robinhood uses an iterative approach to collect event data from metadata servers, where it queries each Metadata Server (MDS) for new events sequentially.

2.4 Metadata Indexing in HPC Storage Systems

Inversion [141] is one of the first systems to propose integrating indexes into the file system. It uses a general-purpose DBMS as the core file system structure, rather than traditional file system inode and data layouts. Brindexer uses file system inode information to build the metadata index database. BeFS [77] uses B+tree to index file system metadata. However, it suffers from scalability issues.

Recent metadata indexing techniques include, Spyglass [111], SmartStore [90], Security Aware Partitioning [146], and GIGA+ [149] which use a spatial tree, such as k-d tree [210], or R-tree [82] to index metadata. Brindexer instead uses RDBMS with 2-level database sharding to efficiently store metadata index information. Other recent metadata indexing tools, GUFI [9], Robinhood Policy Engine [107], and BorgFS [3], use an external database for indexing. The metadata snapshot is taken to an external node where the indexing is performed. Brindexer uses an in-tree design so that no external resources are used that compromises scalability of the indexing approach. PROMES [118] is another recent approach which uses provenance to efficiently improve metadata searching performance in storage systems. However, provenance depends on building relationship graph which is infeasible on large-scale HPC storage systems. Therefore, this technique serves well for single node file systems [18, 65, 66, 151, 152, 153, 154, 155, 159, 160, 161, 162, 163, 179, 196].

Dindex [214] is a distributed indexing technique which comprises hierarchical index layers, each of which is distributed across all nodes. It builds upon distributed hashing, hierarchical aggregation, and composite identification. BRINDEXER uses leveled partitioning technique so that every disjoint sub-tree can be indexed in parallel. TagIt [178], Someta [185], and EMPRESS [106] are metadata management systems that enable "tag and searching". The metadata can be enriched by using custom tags for filtering, pre-processing or automatics metadata extraction. However, this is inbuilt into the storage system. Client nodes have more power and faster interconnect, therefore the indexing tool can leverage that power like BRINDEXER to index metadata from the file system clients [59, 99, 192, 193, 194, 195].

2.5 Load Management in HPC Storage Systems

Load management has been incorporated into a number of modern distributed storage system designs. GlusterFS [46] uses elastic hashing algorithm that completely eliminates location metadata to reduce the risk of data loss, data corruption, and data unavailability. However, no load balancing is supported across the storage targets. Ceph [135, 201] uses dynamic load balancing based on CRUSH [202], a pseudo-random placement function. It also adds limited support for read shedding, where clients belonging to a read flash crowd are redirected to replicas of the primary copy of the data.

Several recent storage systems have explored optimization techniques for load balancing. In [68], dynamic data migration is proposed to balance the load under various constraints. Such approaches add the overhead of migration, while also maintaining availability and consistency. The VectorDot [180] algorithm is able to incorporate all these different constraints, as it is a multidimensional knapsack problem. It is suitable for hierarchical storage systems, as it can model hierarchical constraints [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 49, 54, 55, 60, 70, 100, 101, 117, 177, 182, 186, 197, 207, 220, 221, 222, 224].

Machine learning and data mining techniques have also been used for the more general problem of resource allocation that also includes some load balancing. Martinez et. al. [129] introduce basic learning techniques for improving scheduling in hardware systems. These techniques are focused on individual hardware components and cannot be easily adapted to distributed file systems. A rule based approach to balance load in distributed file servers using graph mining methods is proposed in [78], where access patterns of files is used to relocate the file sets among different file servers. Schaerf et. al. [171] explore the problem space of adaptive load balancing using reinforcement learning techniques. Game Theory is also used for resource allocation [161]. Google has recently explored machine learning to optimize various system-level metrics [132].

Chapter 3

Understanding HPC Application I/O Behavior Using System Level Statistics

3.1 Introduction

The current trend for high performance computing (HPC) systems is that processor performance improves at a rate of 20% per year, while disk access time improves by only 10% every year [85]. As a result, massively parallel HPC applications can suffer from imbalance in computation and I/O performance, with I/O operations becoming a limiting factor in application efficiency [105]. To mitigate this problem, much effort has been dedicated to implementing high performance parallel file systems to support the I/O needs of HPC applications. The Lustre file system [142] is one of the most widely-used parallel file systems, supporting seven of the top ten supercomputers in the latest Top-500 list (June, 2020) [72].

Even though HPC parallel file systems such as Lustre provide much higher bandwidth and reliability than other options, continual improvement of parallel file systems is needed to reduce the impact of the increasing I/O performance bottleneck. File system designers need a comprehensive understanding of the I/O workloads on current HPC systems so that they can design successful next-generation file systems. Additionally, HPC system designers and system administrators need to understand both file system and application behavior so that they can design and tune HPC systems to run as efficiently as possible. Traditionally, I/O and storage analysis efforts have concentrated on analyzing application I/O behavior from application-level statistics [88, 89, 97, 122, 139, 147, 169], and there have been significant studies on the I/O workloads of large scale systems [50, 51, 52, 105, 123, 199, 200] which identify potential I/O bottlenecks in applications and suggest improvements to HPC users. However, while these studies can give clues about how particular applications utilize and stress HPC file systems, they do not give true insight into storage system performance under a general load. Thus, in order to draw meaningful conclusions about how to improve parallel file system designs for all users of an HPC system, we need to analyze statistics captured from the file system itself (from file system data on compute nodes and from data on servers that manage the I/O requests from HPC applications) independently of user applications.

Some of the earliest studies of HPC file systems utilizing system statistics were in 2010 [175, 225]. These works analyzed the deployment of Lustre file systems and lessons learned from them. However, to the best of our knowledge, there have not been efforts which analyze file system statistics in an application-agnostic manner to understand how to support a general HPC workload. Lawrence Livermore National Laboratory (LLNL) is home to a variety of clusters that utilize Lustre file systems as primary storage, and millions of jobs run on these systems. In this effort, we take advantage of the Lustre resources at LLNL and perform a study of general application behavior using file system statistics from Lustre.

In this chapter, we collect and study file system statistics from two Livermore Computing systems with 15 PiB Lustre file systems at LLNL, namely `Quartz` and `Cab`[1]. We collect two types of data from these systems.

- *Aggregate Job Statistics:* This data represents aggregate statistics collected from file system daemons on compute nodes for all jobs that ran on these two systems during the logging period: for Cab April 2015 – March 2018, and for Quartz April 2017 – March 2018.
- *Time-Series Job Statistics:* This data represents time series data collected at 60-second intervals from the metadata server and object storage servers of Lustre for each job; i.e., for each job, we record summary statistics for every minute of the running job during which I/O operations occurred. We collected this data from Quartz for the period June 7, 2018 – July 10, 2018.

We analyze these system statistics without knowledge of the types of applications running on these systems with the goal of answering questions such as:

- What are the typical I/O characteristics of I/O-heavy jobs? For example, do the jobs perform efficient writes with large byte counts per operation? Or do they perform many small (inefficient) write requests?
- How do I/O operations of jobs affect the metadata server?
- Is I/O traffic heavier on any particular day of the month or day of the week?
- Can a single long-running job have a significant affect on the performance of the object storage servers?

[1] Cab was decommissioned in June 2018.

- How does the size of application output files correlate with compute node memory size?

Our study of application-agnostic I/O statistics from the Lustre file system contributes to the state-of-the-art in I/O behavior understanding. We provide insight into how general HPC workloads affect the performance of file systems, which can aid system architects in improving file system and storage system designs and system administrators in tuning existing systems and advising users of best practices. These improvements will help to alleviate the I/O imbalance in HPC systems and increase the overall efficiency of HPC applications.

3.2 Background

In this section, we first describe the architecture of the Lustre file system. Following this, we describe LLNL computing systems we utilized in this work.

3.2.1 Lustre Parallel File System

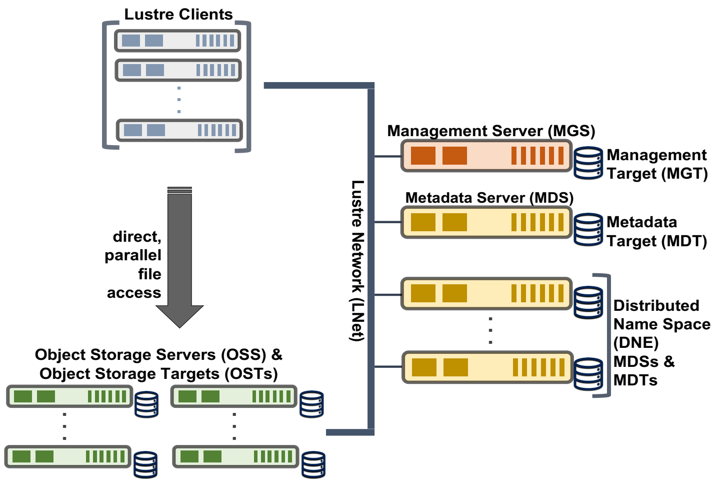

Lustre Clients
direct, parallel file access
Object Storage Servers (OSS) & Object Storage Targets (OSTs)
Lustre Network (LNet)
Management Server (MGS)
Management Target (MGT)
Metadata Server (MDS)
Metadata Target (MDT)
Distributed Name Space (DNE) MDSs & MDTs

Figure 3.1: An overview of Lustre architecture.

The architecture of the Lustre file system is shown in Figure 3.1. Lustre has a client-server network architecture and is designed for high performance and scalability. The *Management Server (MGS)* is responsible for storing the configuration information for the entire Lustre file system. This persistent information is stored on the *Management Target (MGT)*. The *Metadata Server (MDS)* manages all the namespace operations for the file system. The namespace metadata, such as directories, file names, file layout, and access permissions are stored in an *Metadata Target (MDT)*. Every Lustre file system must have a minimum of one MDT. *Object Storage Servers (OSSes)* provide the storage for the file contents in a Lustre file system. Each file is stored on one or more *Object Storage Target (OST)*s mounted on the OSS. Applications access the file system data via *Lustre clients* which interact with OSSes directly for parallel file accesses. The internal high-speed data networking protocol for Lustre file system is abstracted and is managed by the *Lustre Network (LNet)* layer.

3.2.2 Clusters

Table 3.1: Cluster Configurations.

	Cab	**Quartz**
Processor Architecture	Xeon 8-core E5-2670	Xeon 8-core E5-2695
Operating System	TOSS 2	TOSS 3
Processor Clock Rate	2.6 GHz	2.1 GHz
Nodes	1,296	2,634
Cores per node	16	36
Total Cores	20,736	96,768
Memory per node	32 GB	128 GB
Total Memory	41.5 TB	344.06 TB
Interconnect	QDR Infiniband	Intel Omni-Path 100 Gb/s
Tflops	426.0	3,251.4

Table 3.1 gives an overview of the two compute clusters (`Cab` and `Quartz`) at LLNL used in our study. Both clusters have a 15 PiB Lustre file system as primary storage.

3.3 Data Collection

We collected two categories of file system data from `Cab` and `Quartz`.

- Aggregate Job Statistics
- Time-Series Job Statistics

3.3.1 Aggregate Job Statistics

The aggregate job statistics were collected on the client (compute) nodes, by gathering Lustre counter data just before and after the job runs, and calculating the difference. The counter data are acquired from */proc/fs/lustre/llite/lustre-file-system/stats* which is exported by the Lustre client. The statistics in this file reflect the requests as they pass through the interface between the Linux Virtual File System (VFS) and Lustre. Any file system request handled entirely by VFS is not included, nor is a background activity such as re-transmission of a low-level Lustre (not user process) request after server recovery from a crash. All the read- and write-related system calls result in VFS calling into Lustre code, so the read- and write-related counters we use reflect every system call the job made. The statistics are per file system, not per Lustre OSS. These counters start at 0 when the Lustre file system is mounted, and are monotonically increasing until they wrap at $2^{63} - 1$.

The specific statistics used are:

- *starttime, endtime, duration, uid, nodes*: These give the time when the job was started, when it ended, the duration of the job, the anonymized user ID of the job submitter, and the number of nodes on which the job ran.
- *mkdir, mknod, open, rename, rmdir, unlink*: These are the total metadata statistics recorded for the job.
- *read_bytes, write_bytes*: These represent the total number of bytes read and written by the job to the Lustre file system.
- *read_bytes_count, write_bytes_count*: These give the number of read and write calls made by the job to the Lustre file system.
- *recv_bytes, recv_count, send_bytes, send_count*: These network statistics give the number of packets received and sent as well the number of bytes received and sent over LNet.

Both `Cab` and `Quartz` use the SLURM job scheduler [172]. SLURM was configured to run a prolog script after nodes have been allocated, but before the user's job script is run. This prolog script records the counts from the Lustre procfile (*/proc/fs/lustre/llite/lustre-file-system/stats*) at that time, for each node in the allocation. After the job script completes, slurm runs an epilog script. For each node in the allocation, the epilog script extracts the counters from the procfile, and calculates the difference between the "after" and "before" values for each counter. These per-node totals are then summed to obtain the total for the job. This total is stored in an RDMS database, which we queried for this data collection.

The *Aggregate Job Statistics* were collected on `Cab` for three years (April 2015 – March 2018), whereas on `Quartz`, they were collected for one year (April 2017 – March 2018).

3.3.2 Time-Series Job Statistics

Time series data on Lustre file system usage was gathered on the Lustre server nodes via the Lustre JobStats feature [126]. JobStats includes a job ID in every request the Lustre client sends to the servers. The Lustre server records statistics describing the requests received per Job ID. As a result, file system requests which are handled entirely by VFS, or which are satisfied by cached data on the client node, are not reflected in JobStats data. These statistics are gathered per-server, so the total I/O for a job is the sum of the values reported by all servers. File system `read()` and `write()` related requests include a count of bytes to be transferred. JobStats records the number of such requests received, the minimum and maximum byte count seen in requests received so far, and the sum of bytes transferred. If no requests for a given job ID are received for 600 seconds, statistics for that job ID are discarded. The absence of a job ID in the statistics on a server means the server received no requests for that job within the last 600 seconds, and so is equivalent to 0 valued counters.

We collected data from the servers using Telegraf [93] and a customized lustre2 plugin which samples the statistics by reading */proc* files Lustre exports. The proc files were */proc/fs/lustre/mdt/*/job_stats* on Lustre metadata servers, and */proc/fs/lustre/obdfilter/*/job_stats* on Lustre object storage servers. We took one sample every 60 seconds. The data gathered by Telegraf was stored in influxdb [92]. The raw samples were dumped from influxdb as CSV for analysis.

The specific statistics used are:

- ***MDS*** - *jobstats_create, jobstats_mkdir, jobstats_mknod, jobstats_open, jobstats_rename, jobstats_rmdir, jobstats_unlink*
- ***OSS*** - *jobstats_read_bytes, jobstats_read_calls, jobstats_write_bytes, jobstats_write_calls*
- *jobid, time* - Every statistic has a job ID and timestamp attached to it.

Time-Series Job Statistics were collected on `Quartz` for a period of 34 days (June 7, 2018 – July 10, 2018).

3.4 Analysis

We first analyze the aggregate job statistics and see the trends of jobs submitted to both Cab and Quartz. Then we analyze the time-series job statistics to know the details of job runs.

3.4.1 Aggregate Job Statistics

Overview of Jobs

On *Cab*, the aggregate job statistics were collected for a period of three years (April 2015 to March 2018). *2,854,478* total jobs ran during this period. The number of unique users which ran the jobs is *994*.

On *Quartz*, the aggregate job statistics were collected for a year (April 2017 to March 2018). *1,401,897* jobs ran during the one year and there were *584* unique users.

We divide these jobs into two groups:

- `DAT jobs`: Jobs which need to run for more than 24 hours require a special Dedicated Application Time (DAT) request. Users receive expedited access to the systems that would allow them to run larger problems for longer periods of time. On *Cab*, the number of DAT jobs over three years was *1,868* which were run by *48* unique users. On *Quartz*, in one year, the number of DAT jobs was *4,729* an the number of unique users running these jobs was *134*.
- `Non-DAT jobs`: These were the jobs which ran for less than 24 hours. Our extended analysis focuses mostly on non-DAT jobs as these were the most common jobs submitted to the clusters.

Table 3.2: Overview of the Jobs run on Cab and Quartz.

	Cab	**Quartz**
Duration of Data Collection	Apr'15 - Mar'18	Apr'17 - Mar'18
Total Number of Jobs	2,854,478	1,401,897
Total Number of Unique Users	994	584
Number of Dedicated Application Time (DAT) Jobs	1,868	4,729
Number of Unique Users Running DAT Jobs	48	134
#Users with DAT Jobs Doing No I/O to Lustre	10	20
#Users with non-DAT Jobs Doing No I/O to Lustre	196	113

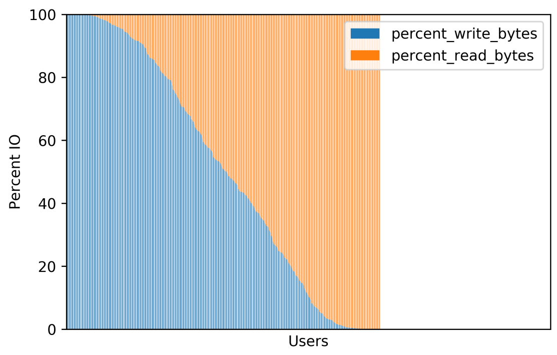
Table 3.2 summarizes the overview of the jobs. It is seen that less than 0.4% jobs receive expedited access to the systems and run for more than a day.

Overall I/O Statistics

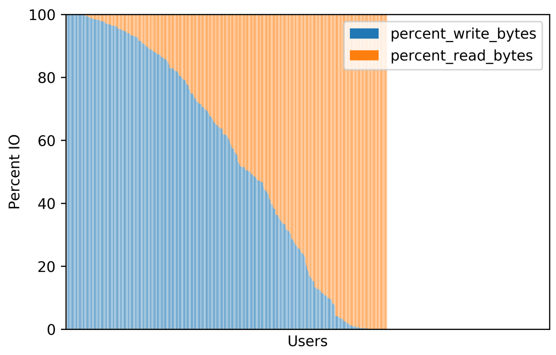
We group both DAT and non-DAT jobs by users and analyze the read and write percentage of jobs run by them. This is shown in Figure 3.2. We find that some users run purely write-intensive workloads and some users run only read-intensive workloads.

> ***Observation 1****: We observe that read-intensive and write-intensive jobs are distributed evenly across users. Previously, a lot of work has been done to optimize performance for write-intensive workloads. However, with the increase in machine learning workloads, which are predominantly read-intensive, there has been an overall rise in number of read-intensive workloads. Therefore, there should be an equal focus in optimizing both reads and writes in a parallel file system.*

Therefore, there should be an equal focus in optimizing both reads and writes in a parallel file system.



(a) Cab: Percentage I/O by user.

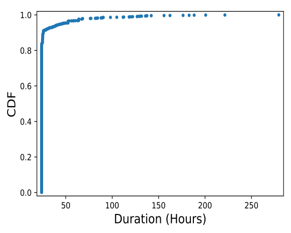


(b) Quartz: Percentage I/O by user.

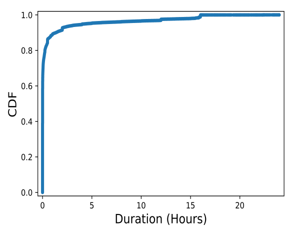
Figure 3.2: Read and write percentage of users on Cab and Quartz.

Analysis of Duration and Number of Nodes

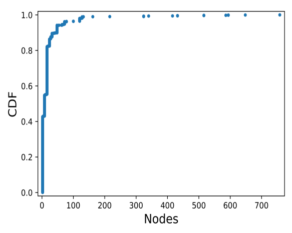
To gain insight into the behavior of jobs in terms of their duration and the number of nodes used, we plot Cumulative Distribution Function (CDF) for both of these metrics for Cab and Quartz in Figure 3.3.



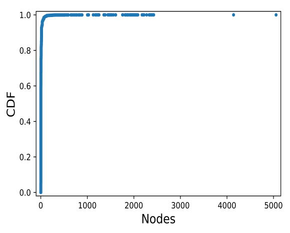
(a) Cab: DAT (CDF Duration)



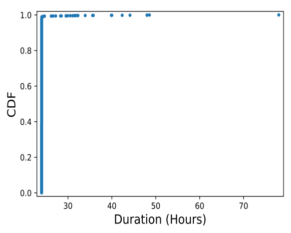
(b) Cab: non-DAT (CDF Duration)



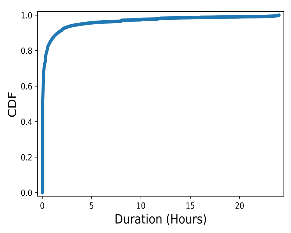
(c) Cab: DAT (CDF #Nodes)



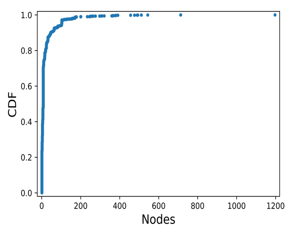
(d) Cab: non-DAT (CDF #Nodes)



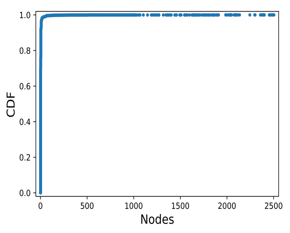
(e) Quartz: DAT (CDF Duration)



(f) Quartz: non-DAT (CDF Duration)



(g) Quartz: DAT (CDF #Nodes)



(h) Quartz: non-DAT (CDF #Nodes)

Figure 3.3: Cumulative Distribution Function for Duration and #Nodes in Cab and Quartz.

- `DAT Jobs:` As can be seen from Figures 3.3a and 3.3e, more than 90% of DAT jobs take between 25 – 30 hours to run on both Cab and Quartz. Upon further analysis of the jobs which run for more than 150 hours on Cab, we found that those jobs did not perform any I/O. For number of nodes used, Figures 3.3c and 3.3g show that 90% of the jobs use less than 100 nodes.
- `non-DAT Jobs:` For the duration of non-DAT jobs, 90% of the jobs run for less 2 hours (Figures 3.3b and 3.3f). The CDF for allocation of nodes of non-DAT jobs shows that 90% of the jobs were allocated less than 100 nodes.

Table 3.3 shows the detailed statistics for duration and number of nodes allocated.

Job Type	Metric	Cluster	min	max	mean	median
DAT	Duration (hours)	Cab	24	279.4	27.6	24
		Quartz	24	78	24.1	24
	#Nodes	Cab	2	758	19.7	8
		Quartz	1	1197	19.8	8
non-DAT	Duration (hours)	Cab	0.0003	23.9	0.8	0.003
		Quartz	0.0003	23.9	0.85	0.03
	#Nodes	Cab	0	5056	8	2
		Quartz	0	2504	3.6	1

Table 3.3: Statistics for duration and nodes for all jobs.

Observation 2*: A huge effort has already been given in HPC storage systems to optimize I/O performance for long running jobs [145, 148]. However, we see that the majority of jobs on a representative real-world system consist of short-running jobs which do not occupy a lot of nodes on the system. Therefore, there should be an equal effort to optimize the I/O performance of small jobs.*

Jobs and Users with Efficient and Inefficient Writes

Starting from this section, we only analyze non-DAT jobs. Jobs with inefficient writes are jobs which write a large amount of data but whose number of bytes per write call is very low. This write pattern reduces I/O performance. We calculate the mean of bytes written by all jobs performing I/O. We also calculate the mean value of bytes written per call for all the jobs performing I/O.

Jobs with inefficient writes have total bytes written greater than the mean total write bytes across all jobs and bytes written per call is less than the mean value of writes per call across all jobs. Jobs with efficient writes have both write bytes as well as the bytes written per write call greater than the respective mean values across all jobs.

- *Jobs with inefficient writes:* (bytes written > mean bytes written) and (bytes written per call < mean bytes written per call)

- *Jobs with efficient writes:* (bytes written > mean bytes written) and (bytes written per call > mean bytes written per call)

We first see how many jobs had inefficient writes and then we focus on the users by grouping jobs by user IDs. The statistics for write bytes and write bytes per call on both Cab and Quartz are shown in Table 3.4.

Classification	**Metric**	**Cluster**	**min**	**max**	**mean**	**median**
Jobs	Write bytes	Cab	81 KB	474 TB	14.4 GB	39 GB
		Quartz	40 Bytes	597 TB	18.1 GB	146 GB
	Write bytes per call	Cab	1 Byte	820 MB	127 KB	390.9 KB
		Quartz	1.0 Byte	1.6 GB	219 KB	11.5 MB
Users	Write bytes	Cab	906 KB	4.8 PB	41.4 TB	7.4 TB
		Quartz	424 KB	4 PB	43.4 TB	27.1 TB
	Write bytes per call	Cab	178 KB	30.5 GB	184.7 MB	123.9 MB
		Quartz	899 KB	108.9 GB	350.4 MB	86.2 MB

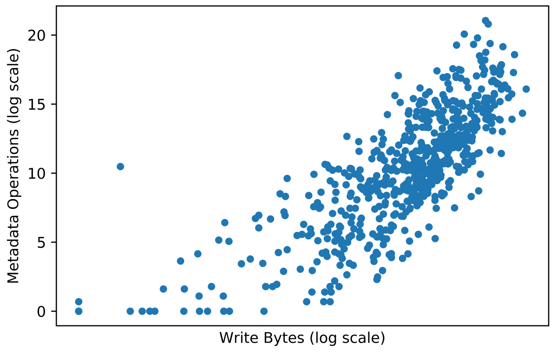
Table 3.4: Statistics for write bytes and write bytes per call for all jobs and users performing I/O on Cab and Quartz.

- *Classification by Jobs:* On Cab, out of 2,563,299 jobs which write more bytes than the mean value, 1,654,938 jobs (64.6%) had inefficient writes. On Quartz, out of 1,295,473 jobs, the number of jobs with inefficient writes was 893,462 (69%). The number of jobs with efficient writes on Cab and Quartz were 869,046 and 277,911 respectively.
- *Classification by Users:* On Cab, out of the 294 users whose jobs write more than the mean value of write bytes, the number of users performing inefficient writes was 138 (46.9%) and the number of users performing efficient writes was 62 (21%). On Quartz, the number of users performing inefficient and efficient writes were 111 (66%) and 57 (34%) users respectively out of 168 users who write more than the mean value of write bytes.

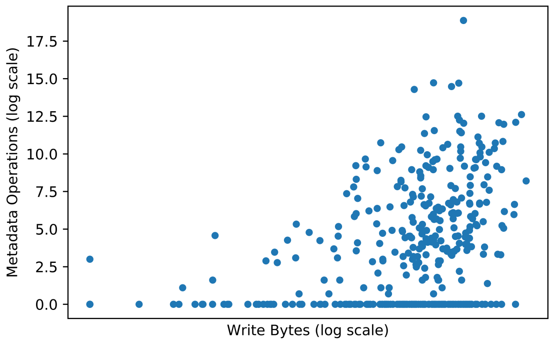
Observation 3*: The number of jobs and correspondingly the number of users who perform efficient writes is very less. Therefore, HPC application developers should be trained to write optimal number of bytes per write call so that the applications can achieve better I/O performance.*

Metadata Operations and Write Bytes

The metadata operations we consider are *mkdir* and *mknod*. We sum all metadata operations in a job and compare the sum with the write bytes for that job. The log-scale values are plotted and are shown in Figure 3.4. In both Cab and Quartz, it is seen that the number of metadata operations become larger for larger number of bytes written. We computed the



(a) Cab: Metadata vs Write



(b) Quartz: Metadata vs Write

Figure 3.4: Relationship between Write Bytes and Metadata operations for Jobs in Cab and Quartz.

correlation coefficient, R, for each and found that $R = 0.93$ for Cab and $R = 0.59$ for Quartz, indicating strong and moderate correlations, respectively.

Observation 4*: There is a positive correlation between metadata operations and writes, especifically for mkdir and mknod. Therefore, to improve the I/O performance, metadata operations need to be handled carefully by the metadata servers.*

Behavior of Metadata Servers

To manage the increase in metadata, Lustre incorporates distributed namespace (DNE) [125] - more than 1 MDTs in large HPC storage systems. Here, Lustre has 14 MDTs. The number of file opens and close requests which are being handled by different MDTs are shown in Figure 3.5.

It is seen in Figure 3.5 that the number of file opens that being handled is consistent with the number of jobs and is expected. However, we observe that not all files which are opened are closed. Moreover, the number of file open requests handled by different MDTs are different.

Observation 5*: Not all files which are opened are closed which can lead to sub-optimal metadata performance due to dangling file pointers. Therefore, HPC users should be trained to always close open files. Additionally, there is a scope of working on balancing the metadata load across metadata targets for improving the overall I/O performance.*

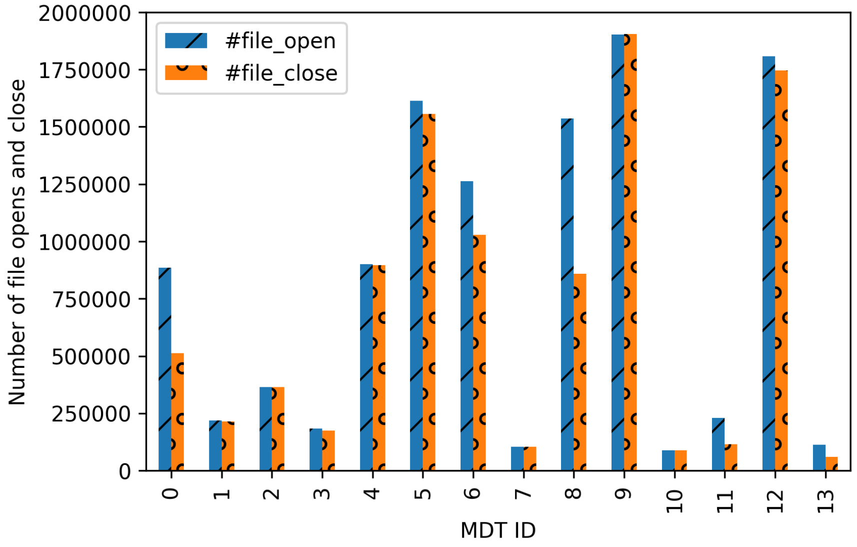
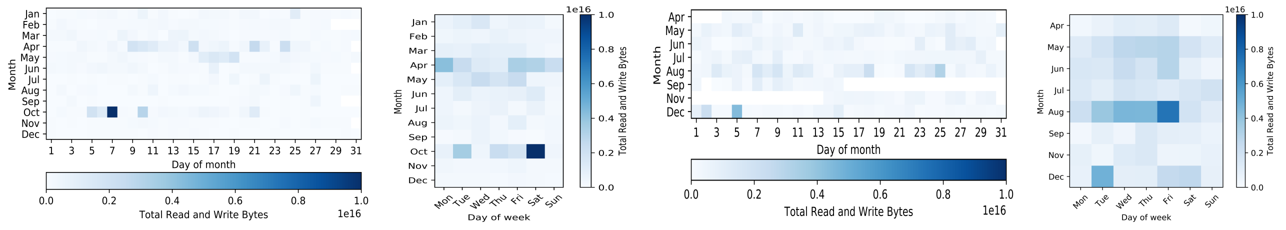


Figure 3.5: #File opens and close handled by different MDTs .

Analysis of I/O Traffic over Time

For this analysis, we asked three questions of our dataset.

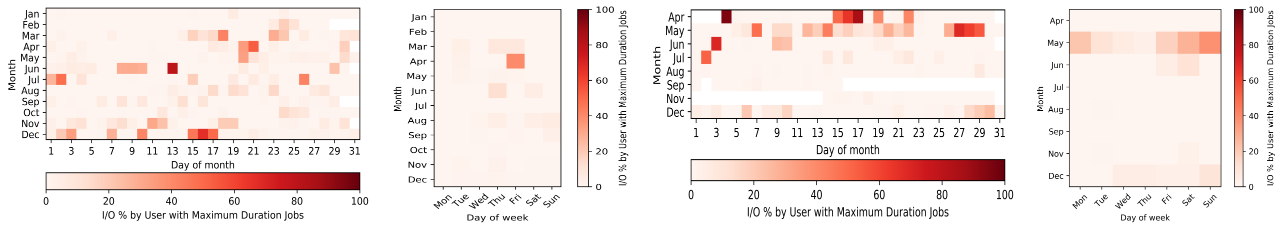
1. Is there any trend in I/O activity for particular months, days of the month, or days of the week?
Are more I/O intensive jobs run during the weekend? Do people run less number of jobs on holidays? To answer these questions, we plotted heat maps showing the I/O traffic over the whole data collection period. Due to space constraints, we only show the year 2017 in Figure 3.6. As seen in the heat maps, there was no particular trend shown by I/O traffic. Therefore, I/O traffic from jobs cannot be predicted by the job start time with respect to the calendar or holidays.



(a) Cab: I/O Per Day
(b) Cab: I/O Per Day of Week
(c) Quartz: I/O Per Day
(d) Quartz: I/O Per Day of Week

Figure 3.6: I/O Heat Map for 2017 per day and per day of the week in Cab and Quartz.

2. How much does a job which runs for the maximum duration affect the overall I/O traffic in a month, day, or a day of the week?
We explored whether a job which ran for a long period tended to be responsible for a large portion of the I/O traffic in a system. We chose the jobs which ran for the longest period of time in a day, or a day of the week. We then calculated the percentage of total system I/O traffic contributed by each of those jobs. Figure 3.7 shows the contribution of a job which runs for the maximum duration to the overall traffic in a particular day and day of the week in the year 2017. There are few days where the contribution is significant, but overall, no significant affect was seen.

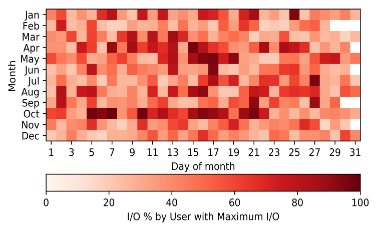


(a) Cab: Per Day (b) Cab: Per Day of Week (c) Quartz: Per Day (d) Quartz: Per Day of Week

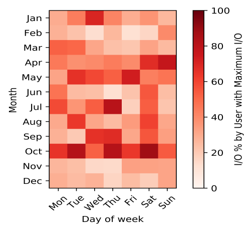
Figure 3.7: Contribution of Job with Maximum Duration on I/O for a day and a day of the week in 2017.

3. How much does a user with the highest bytes read or written in a day or a day of the week affect the overall I/O traffic?
Our last question was how much does one user's I/O contribute to the overall I/O traffic in the system. To help answer this question, we found the users who performed the maximum I/O (highest bytes read or written) on a particular day or a day of the week. Then, we calculated the percentage of total system I/O these users contributed to that day or day of the week. Figure 3.8 shows the contribution of users with maximum I/O on the overall I/O traffic for the year 2017. As can be seen, these users have a very significant impact on the overall I/O of the system. Therefore, job schedulers ideally would schedule other jobs with little or no I/O while these users run their jobs to reduce I/O contention and job run time.

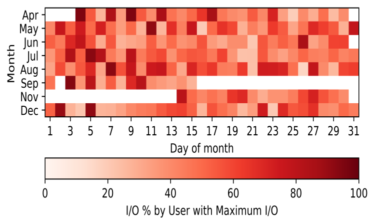
Observation 6*: There is no particular trend of I/O corresponding to a month, day of a month, or day of a week. Therefore, HPC I/O intensive jobs cannot be less I/O contended if submitted during a particular time. As expected, the I/O during a particular time when multiple I/O intensive jobs run on the system, is dominated by the job which does the maximum I/O rather than the job which runs for the maximum duration. Therefore, I/O optimizations should not be focused for long running jobs, and there is no correlation between the duration of a job and the amount of I/O requests that the job generates.*



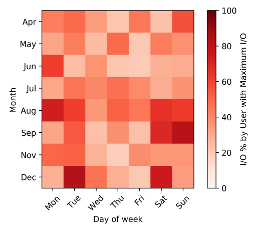
(a) Cab: Per Day



(b) Cab: Per Day of Week



(c) Quartz: Per Day



(d) Quartz: Per Day of Week

Figure 3.8: Contribution of User doing maximum I/O (highest read/write bytes) with respect to total I/O on the system in 2017 on a day and a day of the week's I/O.

3.4.2 Time-Series Job Statistics

The next few analysis sections involves analyzing the time-series job statistics.

Temporal distribution of I/O within jobs' runs

Our analysis shows that more than 53% of the jobs submitted to the clusters perform some I/O. But how are the I/O operations distributed across the jobs' run time?

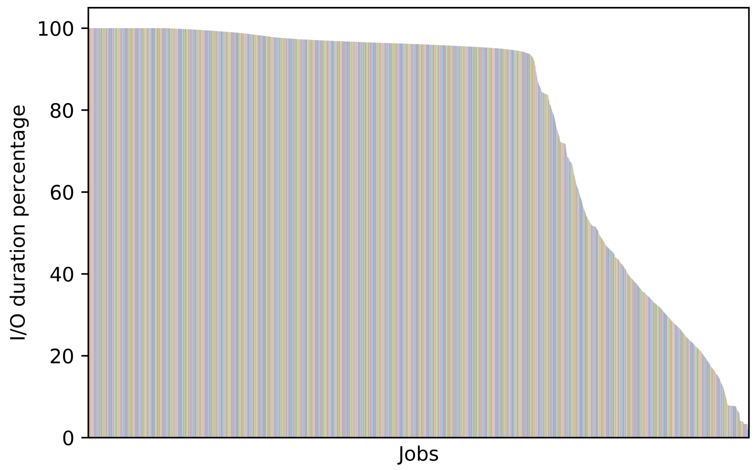


Figure 3.9: Percent minutes 11,425 jobs performed any I/O.

All 16,795 jobs in the time-series dataset performed some I/O (the collection method excludes jobs without I/O). Of those jobs, 11,425 performed either read, write, or both operations. 10,443 jobs performed write operations. Since the time-series dataset records reflect I/O performed during each one-minute span, we can determine what percentage of the minutes during a job's run saw some I/O performed. We will call this percentage "I/O share". Note that during a given minute a single byte written or read causes that minute to be included

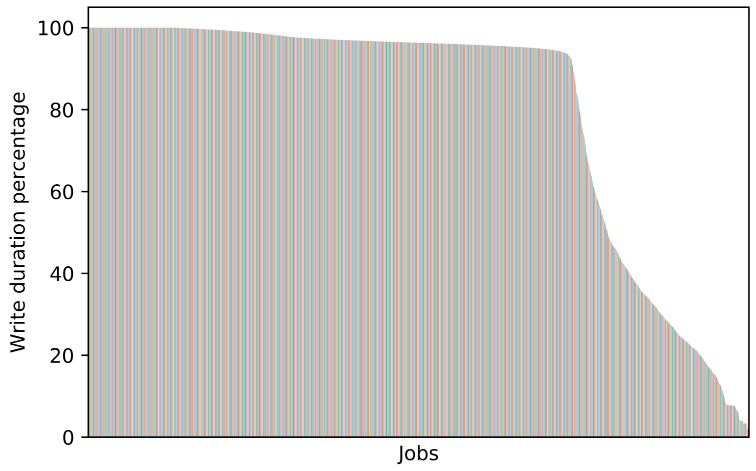


Figure 3.10: Percent minutes 10,443 jobs performed at least one write.

in the I/O share, even though the I/O may have taken only a small fraction of a second. Figure 3.9 shows the I/O share for jobs which performed read, write, or both operations. The mean I/O share for these jobs is 78.8%. I/O share for jobs performing write operations is shown in Figure 3.10. The mean is 80.4%. Both graphs in Figures 3.9 and 3.10 are similar, therefore write operations dominate the I/O performed by jobs.

***Observation 7**: On average, jobs which perform I/O spread I/O activities across 78.8% of their runtime. Therefore, I/O optimizations cannot be focused at only certain time instances of job runtime. We need to work on developing I/O optimizations that aim to lower I/O contention and improve the overall I/O performance for most of the duration of application runtime.*

Relationship between write bytes and memory

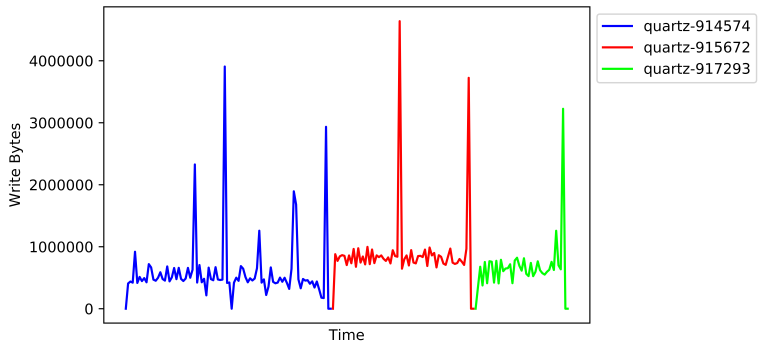


Figure 3.11: Write Bytes written over time by 3 random jobs in Quartz.

Figure 3.11 shows the write pattern for 3 randomly chosen jobs. It is seen that the write bytes over time is periodic. Therefore, we assume that the bursts represent writing a single file and we add up the write bytes to get the size of the file. This single file could represent a checkpoint of the application data.

In the Quartz time-series dataset, 16,795 jobs were recorded. The total memory in one node in Quartz is 128 GB. We calculate the size of memory as 128 GB $*$ job node count. Only 170 jobs wrote in bursts larger than 50% of memory. 16,485 jobs wrote bursts which are smaller than than 15% of memory, while 16,173 jobs wrote bursts smaller than 5% of memory.

Observation 8*: More than 95% of applications write in bursts of size less than 5% memory. Therefore, memory size is not a good predictor of write burst size as is used in many previous I/O optimization works [130]. We need better prediction approaches, such as [96] to predict I/O bursts.*

Next, we inspect the write burst patterns for all I/O jobs in Quartz. First, burst size is compared to memory size. For every job, the I/O duration of the job is divided into 5 categories.

- `percent_LessThan1:` Percent of the total I/O duration when the job writes bursts are $<$ 1% memory.
- `percent_1To5:` %Duration of the total I/O duration when the job writes bursts are $\geq$ 1% and $<$ 5% memory.
- `percent_5To10:` %Duration of the total I/O duration when the job writes bursts are $\geq$ 5% and $<$ 10% memory.
- `percent_10To50:` %Duration of the total I/O duration when the job writes bursts are $\geq$ 10% and $<$ 50% memory.
- `percent_50To100:` %Duration of the total I/O duration when the job writes bursts are $\geq$ 50% and $\leq$ 100% memory.

Figure 3.12 shows how much of the job time is associated with different sizes of write bursts, possibly to checkpoint data. It is clear from the figure that most of the jobs spend 100% of the I/O minutes in utilizing less than 1% memory, while there are few jobs which spend all I/O minutes writing bursts in sizes between 10 – 50% of memory.

Observation 9*: 90% of jobs never write bursts larger than 1% of memory size. Therefore, we can ideally use more portions in the memory of an HPC storage system to increase the prefetching capacity and improve the overall I/O performance.*

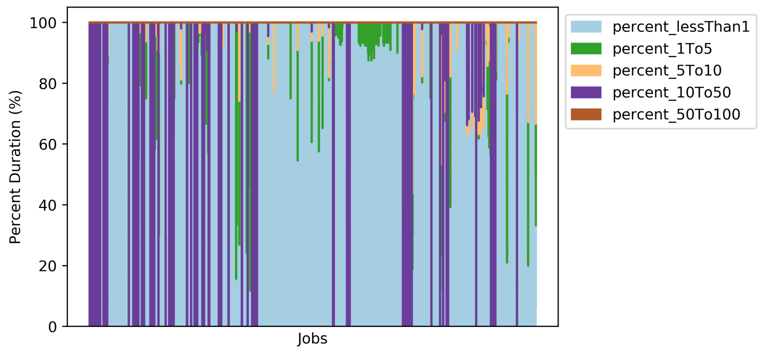


Figure 3.12: % I/O duration vs. write burst size as % of memory.

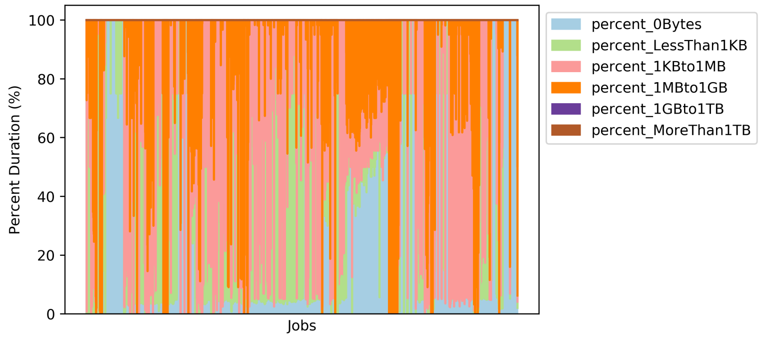


Figure 3.13: Percent I/O duration vs vs size of write bursts.

We also look at the absolute size of the write bursts and what percentage of the job I/O duration is associated with different sizes of bursts. Similar to Figure 3.12, Figure 3.13 shows the percentage of I/O time during which jobs issue different sizes of bursts. This is for all jobs which performed I/O.

The categories of data sizes are:

- `0 Bytes`
- `Less than 1 KB`
- `Between 1 KB and 1 MB`
- `Between 1 MB and 1 GB`
- `Between 1 GB and 1 TB`
- `More than 1 TB`

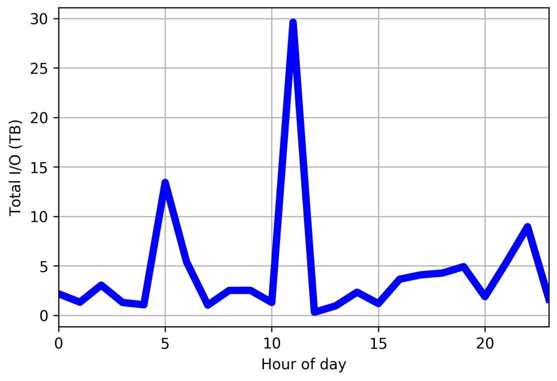
Our analysis shows that 73% jobs spend greater than 50% of their I/O time to write data bursts whose size is between 1 KB and 1 MB. There are many jobs which spend more than 90% time writing bursts of size 1 MB to 1 GB.

***Observation 10**: Most jobs write burst data in the range of few kilobytes for the majority of their I/O duration. In the past, I/O optimizations have worked on large checkpoint data. However, we observe that there is a significant portion of checkpoint data with small writes. Therefore, I/O optimizations should also be done for burst write requests having small number of bytes.*

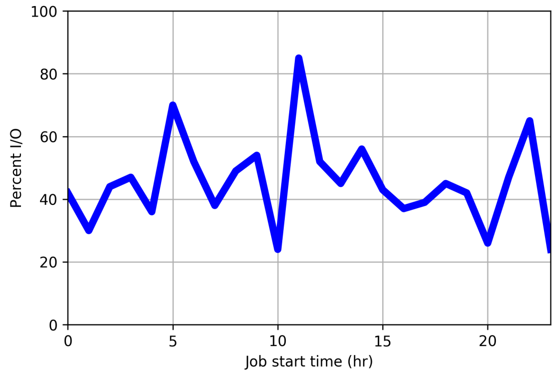
Demystifying I/O Contention

Figure 3.14a shows the total amount of data which is transferred at different hours of the day. We observe that the largest amount of I/O activity is performed by runs which start at 5AM and 11AM local time.

In Figure 3.14b, we plot the percentage of I/O time for applications across different hours of the day. The percentage of I/O time of an application is plotted as percentage of the maximum I/O time among all runs which perform similar I/O behavior to normalize it across applications. However, we observe that runs started at 5AM and 11AM have the highest percentage of I/O time due to the high I/O activity during this time.



(a) Total I/O for every hour throughout the day.



(b) Percentage of I/O time noramlized for similar jobs starting at different hours throughout the day.

Figure 3.14: Plotting I/O behavior of application for every hour throughout the day.

***Observation 11**: I/O contention adversely affects the I/O performance for jobs, resulting in similar applications spending more time doing I/O during the time period when other applications are also performing considerable I/O. This makes I/O performance the bottleneck for improving the overall performance of an HPC application. Therefore, much effort should be done in handling I/O contentions in the HPC centers.*

3.5 Discussion

The analysis of the server level statistics gave interesting insights into HPC application I/O behavior. The study though focused on Livermore Computing resources, the insights can be extended for other HPC centers. The study is conducted on a real-world deployment of Lustre file system, which is one of the most popular parallel file systems used in the top 100 supercomputers [72]. The jobs which are studied in this chapter are representative of other high performance computing centers, like National Energy Research Scientific Computing Center (NERSC) [148], and Oak Ridge Leadership Computing Facility (OLCF) [145].

3.5.1 Lessons for HPC admininistrators

- There is an equal distribution of reads and writes per user in the HPC storage system. Therefore, equal focus needs to be given on optimizing both reads and writes in a parallel file system.
- The mean duration of jobs is less than 52 minutes which suggests that job scheduling and I/O contention strategies should be developed for shorter duration jobs rather than jobs which run for many hours.
- It also seems to contradict the conventional wisdom that defensive I/O is the primary use of HPC file systems, which may explain the very small number of jobs which wrote bursts larger than 1% of memory.
- A small number of jobs generated most of the load on the file system. Focusing on improving the I/O behavior of jobs which perform maximum I/O will be more beneficial for overall I/O performance than focusing on jobs which run for long durations.
- Very few applications perform efficient writes to the file system. There are applications which write burst sizes greater than 50% of memory; these may be checkpoints.
- Effort should be made in identifying periods of I/O contention in different HPC centers and users should be educated to submit I/O intensive jobs outside of those times to improve I/O performance of the overall system.
- There should be optimizations for balancing the metadata load across the metadata servers which can adversely impact the I/O performance.

3.5.2 Lessons for HPC users

- Users who run write-intensive workloads on parallel file system need to perform efficient writes. Only 22% users perform efficient writes, which degrades the I/O performance of the entire system. The users can get better I/O performance by performing more write bytes per write function call.

- Starting an I/O intensive job at particular time instants can be beneficial in facing less I/O contention which would improve the overall performance of the application. Therefore, HPC users should know which periods of the day are bad for submitting I/O intensive applications.
- A lot of memory remains unused for I/O operations. Therefore, HPC application developers can improve I/O performance by prefetching data into memory.

3.6 Chapter Summary

Improving I/O performance has become the most important factor in modern I/O bound HPC applications. Therefore, understanding the I/O behavior of HPC applications is very important for system administrators, file system developers, and HPC users. This chapter collected Lustre file system server level statistics from two clusters, Cab and Quartz, at the Lawrence Livermore National Laboratory, for a period of three years and analyzed the statistics in an application-agnostic manner. Our studies have indicated interesting results which show that due to the increase in popularity of machine learning jobs, the HOC jobs now have an even distribution of write-intensive and read-intensive jobs, showing the importance of giving equal priority in improving file system read and write performance. Our analysis also led us to believe that there should be focus on I/O optimizations for jobs which run for short duration. Also, there should be efforts to educate HPC users to develop applications which perform efficient writes. Moreover, the metadata spread across metadata servers are unbalanced and there should be efforts to balance the load or migrate metadata operations to less loaded metadata server. Much effort is also needed to mitigate the effect of I/O contention that adversely impacts the I/O performance of the entire system. We believe that our analysis will help all HPC practitioners to build better file systems and utilize it more effectively.

Chapter 4

Optimizing Data Partitioning for In-Memory Data Analytics Frameworks

4.1 Introduction

Large scale in-memory data analytic platforms, such as Spark [211, 212], are being increasingly adopted by both academia and industry for processing data for a myriad of applications and data sources. These platforms are able to greatly reduce the amount of disk I/Os by caching the intermediate application data in-memory, and leverage more powerful and flexible direct acyclic graphs (DAG) based task scheduling. Thus, in-memory platforms outperform widely-used MapReduce [67]. The main advantage of a DAG-based programming paradigm is the flexibility it offers to the users in expressing their application requirements. However, the downside is that complicated task scheduling makes identifying application bottlenecks and performance tuning increasingly challenging.

There is a plethora of application-specific parameters that impact runtime performance in Spark, such as tasks parallelism, data compression and executor resource configuration. In typical data processing systems, the input (or intermediate) data is divided into logical subsets, called partitions. Specifically, in Spark, a partition can not be divided between multiple compute nodes for execution, and each compute node in the cluster processes one or more partitions. Moreover, a user can configure the number of partitions and how the data should be partitioned (i.e., hash or range partitioning schemes) for each Spark job. Suboptimal task partitioning or selecting a non-optimal partition scheme can significantly increase workload execution time. For instance, if a partition strategy launches too many tasks within a computation phase, this would lead to CPU and memory resource contention, and thus lose performance. Conversely, if too few tasks are launched, the system would have low resource utilization and would again result in reduced performance.

Moreover, inferior partitioning can lead to serious data skew across tasks, which would eventually result in some tasks taking significantly longer time to complete than others. As data processing frameworks usually employ a global barrier between computation phases, it is critical to have all the tasks in the same phase finish approximately at the same time, so as to avoid stragglers that can hold back otherwise fast-running tasks. The right scheme for data partitioning is the key for extracting high performance from the underlying hardware resources. However, finding a data partitioning scheme that gives the best or highest performance is non-trivial. This is because, data analytic workflows typically involve complex algorithms, e.g., machine learning and graph processing. Thus, the resulting task execution plan can become extremely complicated with increasing number of multiple computation phases. Moreover, given that each computation phase is different, the optimal number of partitions for each phase can also be different, further complicating the problem.

Spark provides two methods for users to control task parallelism. One method is to use a workload specific configuration parameter, $default.parallelism$, which serves as the default number of tasks to use for when the number of partitions is not specified. The second method is to use repartitioning APIs, which allow the users to repartition the data. Spark does not support changing of data parallelism between different computation phases except via manual partitioning within a user program through repartitioning APIs. This is problematic because the optimal number of partitions can be affected by the size of the data. Users would have to change and recompile the program every time they process a different data set. Thus, a clear opportunity for optimization is lost due to the rigid partitioning approach.

In this chapter, we propose Chopper, an auto-partitioning system for Spark[1] that automatically determines the optimal number of partitions for each computation phase during workload execution. Chopper alleviates the need for users to manually tune their workloads to mitigate data skewness and suboptimal task parallelism. Our proposed dynamic data partitioning is challenging due to several reasons. First, since Spark does not support changing tasks parallelism parameters for each computation phase, Chopper would need to design a new interface to enable the envisioned dynamic tuning of task parallelism. Second, the optimal data partitions differ across different computation phases of workloads. Chopper needs to understand application characteristics that affect the task parallelism in order to select an appropriate partitioning strategy. Finally, to adjust the number of tasks, Chopper may introduce additional data repartitioning, which may incur extra data shuffling overhead that has to be mitigated or amortized. Thus, a careful orchestration of the parameters is needed to ensure that Chopper's benefits outweigh the costs.

To address the above challenges, Chopper first modifies Spark to support dynamically changing data partitions through an application specific configuration file. Chopper checks

[1]We use Spark as our evaluation DAG-based data processing framework to implement and showcase the effectiveness of Chopper. We note that the proposed system can be applied to other DAG-based data processing framework, such as Dryad [94].

different numbers of data partitions before scheduling the next computation phase. Information gathering about the application execution is achieved via several lightweight test runs, which are then analyzed to identify task profiles, data skewness, and optimization opportunities. Chopper uses this information along with a heuristic to compute a data repartitioning scheme, which minimizes the data skew, determines the right tasks parallelism for each computation phase, while minimizing the repartitioning overhead.

Specifically, this chapter makes the following contributions.

1. We enable dynamic tuning of task parallelism for each computation phase in DAG-based in-memory analytics platforms such as Spark.
2. We design a heuristic to carefully compute suitable data repartitioning schemes with low repartitioning overhead. Our approach successfully identifies the data skewness and optimization opportunities and adjusts task parallelism automatically to yield higher performance compared to the default static approach.
3. We implement Chopper on top of Spark and evaluate the system to demonstrate its effectiveness. Our experiments demonstrate that Chopper can significantly outperform vanilla Spark by up to 35.2% for the studied workloads.

4.2 Background and Motivation

In this section, we first discuss current data partitioning methodologies in Spark. Next, we present the motivation for our work by studying the performance impact of different number of data partitions on a representative workload, *KMeans* [83].

4.2.1 Spark Data Partitioning

In Spark, data is managed as an easy-to-use memory abstraction called resilient distributed datasets (RDDs) [212]. To process large data in parallel, Spark partitions an RDD into a collection of immutable partitions (*blocks*) across a set of machines. Each machine retains several blocks of an RDD. Spark tasks, with one-to-one relationship with the partitions, are launched on the machine that stores the partitions. Computation is done in the form of RDD actions and transformations, which can be used to capture the lineage of a dataset as a DAG of RDDs, and help in the recreation of an RDD in case of a failure. Such DAGs of RDDs are maintained in a specialized component, DAGScheduler, which schedules the tasks for execution.

Fig. 4.1 shows an overview of the Spark DAGScheduler. The input to DAGScheduler are called jobs (shown as ActiveJob in the figure). Jobs are submitted to the scheduler using a

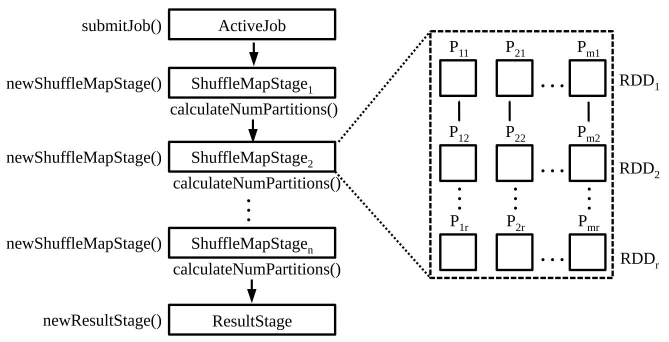


Figure 4.1: Overview of Spark DAGScheduler.

submitJob method. Every job requires computation of multiple stages to produce the final result. Stages are created by shuffle boundaries in the dependency graph, and constitute a set of tasks where each task is a single unit of work executed on a single machine. The narrow dependencies, e.g., *map* and *filter*, allow operations to be executed in parallel and are combined in a single stage. The wide dependencies, e.g., *reduceByKey*, require results to be combined from multiple tasks and cannot be confined to a single stage. Thus, there are two types of stages: ShuffleMapStage (shown in Fig. 4.1 as $ShuffleMapStage_1$, $ShuffleMapStage_2$, ..., $ShuffleMapStage_n$), which writes map output files for a shuffle operation, and ResultStage, i.e., the final stage in a job. ShuffleMapStage and ResultStage are created in the scheduler using *newShuffleMapStage* and *newResultStage* methods, respectively.

In Spark, tasks are generated based on the number of partitions of an input RDD at a particular stage. The same function is executed on every partition of an RDD by each task. In Fig. 4.1, $ShuffleMapStage_2$ is expanded to show the operations involved in a particular stage. Different operations in a stage form different RDDs (RDD_1, RDD_2, ... , RDD_r). Each RDD consists of a number of tasks that can be operated in parallel. In the figure, $P_{\alpha\beta}$ represents $partition_\alpha$ of RDD_β. There are m partitions for an RDD. Each RDD in a stage consists of a narrow dependency on the previous RDD, which enables parallel execution of multiple tasks. Thus, the number of partitions at each stage determins the level of parallelism.

Currently, Spark automatically determines the number of partitions based on the dataset size and the cluster setups. However, the number of partitions can also be configured manually using `spark.default.parallelism` parameter. In order to partition the data, Spark provides two data partitioning schemes, namely hash partitioner and range partitioner. Hash partitioner divides RDD based on the hash values of keys in each record. Data with the same hash keys are assigned to the same partition. Conversely, range partitioner divides a dataset into approximately equal-sized partitions, each of which contains data with keys within a specific range. Range partitioner provides better workload balance, while hash partitioner ensures that the related records are in the same partition. Hash partitioner is the default partitioner, however, users can opt to use their own partitioner by extending the *Partitioner* interface.

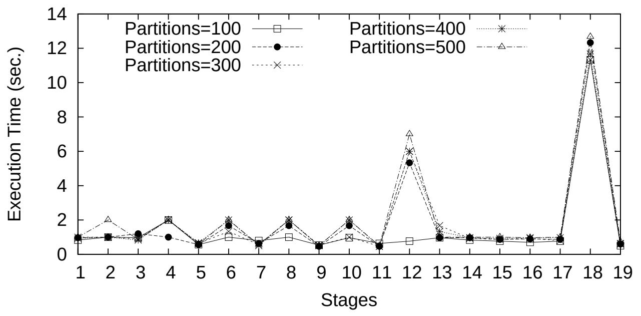


Figure 4.2: Execution time per stage under different number of partitions.

Although Spark provides mechanisms to automatically determine the number of partitions for a given RDD, it lacks application related knowledge to determine the best parallelism for a specific job. Moreover, the default hash partitioner is prone to creating workload imbalance for some input data. Spark provides the flexibility to tailor the configurations for workloads, however, it is not easy for users to determine the best configurations for each stage of a workload, especially when workloads may contain hundreds of stages.

4.2.2 Workload Study

To show the impact of data partitioning on application performance, we conduct a study using *KMeans* workload from SparkBench [112] and the latest release of Spark (version 1.6.1), with Hadoop (version 2.6) providing the HDFS [176] storage layer. Our experiments execute on a 6-node heterogeneous cluster. Three nodes (A, B, C) have 32 cores, 2.0 GHz AMD processors, 64 GB memory, and are connected through a 10 Gbps Ethernet interconnect. Two nodes (D, E) have 8 cores, 2.3 GHz Intel processors, 48 GB memory. The the remaining node (F) has 8 cores, 2.5 GHz Intel processor, and 64 GB memory. Nodes D, E and F are connected via a 1 Gbps Ethernet interconnect. Node F is configured to be the master node, while nodes A to E are worker nodes for both HDFS and Spark. Every worker node has one executor with 40 GB memory, and the remaining memory can be used for the OS buffer and HDFS data node operations. While our cluster hardware is heterogeneous, we configure each executor with the same amount of resources, essentially providing same resources to each component to better match with Spark's needs, and alleviating the performance impact due to the heterogeneity of the hardware. Note that, given the increasing heterogeneity in modern clusters, we do take the heterogeneity of cluster resource into account when designing Chopper. We repeated each experiment 3 times, and report the average results in the following.

First, we study the performance impact of different number of partitions. For this test, we use KMeans workload with 7.3 *GB* input data size. KMeans has 20 stages in total, and we change the number of partitions from 100 to 500 and record the execution time for each stage. Fig. 4.2 shows the results. For every stage, the number of partitions that yields minimum

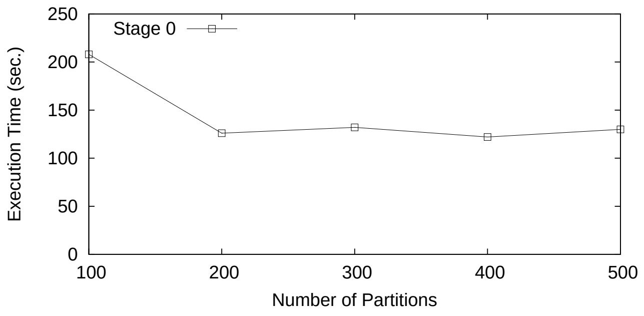


Figure 4.3: Execution time of stage 0 under different partition numbers.

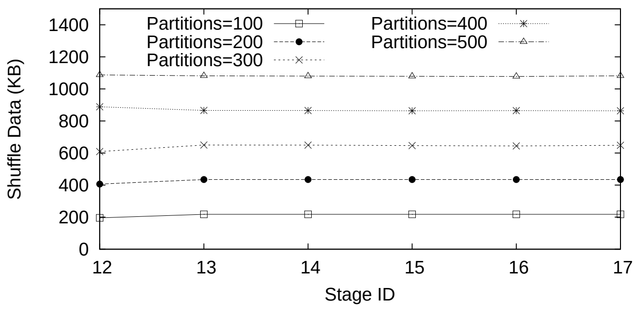


Figure 4.4: Shuffle data per stage under different partition numbers.

execution time varies. This shows that different stages have different characteristics and that the execution time for each stage can vary even under the same configuration. To further investigate the impact on performance, we study stage-0 in more detail. As shown in Fig. 4.3, the execution time of a stage changes with the number of partitions, and we see the worst performance when the number of partitions is set to 100. From this study, we observed that the number of partitions has an impact on the overall performance of a workload. Furthermore, different stages inside a workload can have different optimal number of partitions. In this example, specifying 100 partitions may be an optimal configuration for overall execution (Fig. 4.2), but clearly it is not optimal for stage-0 (Fig. 4.3).

To better understand the above observed performance impact, we investigate the amount of shuffle data produced at each stage with different number of partitions—since shuffle has a big impact on workload performance. For this test, we record the maximum of shuffle read or write data as representative of shuffle data per stage. For KMeans, only stages 12-17 involve data shuffle. As shown in Fig. 4.4, any increase in the number of partitions also increases the shuffle data at each stage. A shuffle stage usually involves repartitioning RDD data. For an equivalent execution time, if repartitioning is not involved, then the amount of shuffle data increases from 434.83 KB for 200 partitions to 1081.6 KB for 500 partitions for stage-17, compared to 217.33 KB of shuffle data when repartitioning is done. Also, when compared to a large number of partitions, e.g., 2000, there is a significant increase in the execution time as well as increase in the amount of shuffle data. For 2000 partitions, the execution

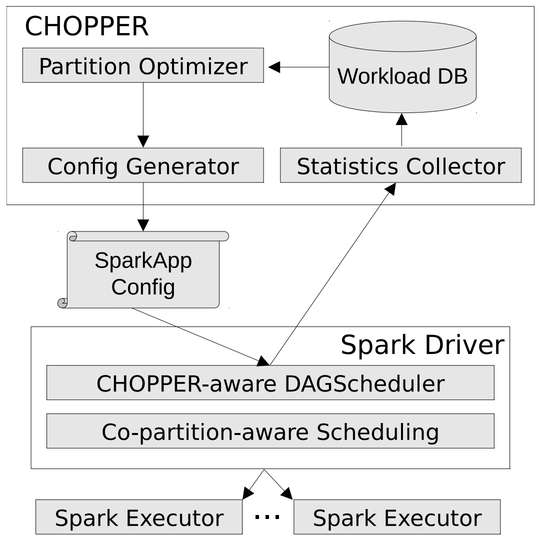


Figure 4.5: System architecture of CHOPPER.

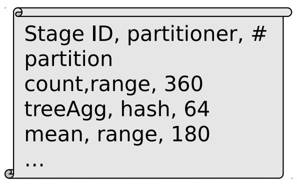
time is 4.53 minutes and the amount of shuffle data for stage-17 is 4300.8 KB. We observe 38.8% improvement in execution time from repartitioning for similar amount of shuffle data. Also there is 46.1% improvement in execution time, and 94.9% reduction in the amount of shuffle data per stage when compared to large (i.e., 2000) number of partitions.

These experiments show that the number of partitions is an important configuration parameter in Spark and can help improve the performance of a workload. The optimal number of partitions not only varies with workload characteristics, but is also different among different stages of a workload. We leverage these findings in designing the auto-partitioning scheme of CHOPPER.

4.3 System Design

In this section, we present the design of CHOPPER, and how it achieves automatic repartitioning of RDDs for improved performance and system efficiency.

Fig. 4.5 illustrates the overall architecture of CHOPPER. We design and implement CHOPPER as an independent component outside of Spark. As Spark is a fast evolving system, we keep the changes to Spark for enabling our dynamic partitioning to a minimum. This reduces code maintenance overhead while ensuring adoption of CHOPPER for real-world use cases. CHOPPER consists of a partition optimizer, a configuration generator, a statistics collector, and a workload database. In addition, CHOPPER extends the Spark's DAGScheduler to support dynamic partitioning configuration and employs a co-partition-aware scheduling mechanism to reduce network traffic whenever possible. *Statistics collector* communicates with Spark to gather runtime information and statistics of Spark applications. The collector can be easily



Stage ID, partitioner, # partition
count,range, 360
treeAgg, hash, 64
mean, range, 180
...

Figure 4.6: An example generated Spark workload configuration in Chopper.

extended to gather additional information as needed. *Workload DB* stores the observed information including the input and intermediate data size, the number of stages, the number of tasks per stage, and the resource utilization information. *Partition optimizer* retrieves application statistics, trains models, and computes an optimized partition scheme based on the current statistics and the trained models for the workload to be optimized. This information is also stored in *Workload DB* for future use. *Partition optimizer* then generates a workload specific configuration file. The extended dynamic partitioning DAGScheduler changes the number of partitions and the partition scheme per stage according to the generated Spark configuration file. Finally, the co-partitioning-aware component schedules partitions that are in the same key range on the same machine if possible to decrease the amount of shuffle data. The partition optimizer does not need to consider data locality because the input of the repartitioning phase is the local output of the previous Map phase, and the destinations of the output of the repartitioning phase depend on the designated shuffle scheme. Thus, existing locality is automatically preserved.

Our system allows dynamic updates to the Spark configuration file whenever more runtime information is obtained. Chopper modifies Spark to allow applications to recognize, read, and adopt the new partition scheme.

4.3.1 Enable Auto-Partitioning

An example application configuration file produced by Chopper is shown in Fig. 4.6. It consists of multiple tuples each containing a signature; the partitioner, and the number of partitions for a particular stage. We use stage signatures to identify stages that invoke identical transformations and actions. This is helpful when the number of iterations within a machine learning workload is unknown, where we can: (a) use the same partition scheme for all the iterations; or (b) use previously trained models to dynamically determine the number of partitions to use if the intermediate data size changes across iterations.

When an application is submitted to a Spark cluster, a Spark driver program is launched in which a SparkContext is instantiated. SparkContext then initiates our auto-partitioning aware DAGScheduler. The scheduler checks the Spark configuration file before a stage is executed. If the partition scheme is different from the current one, the scheduler changes the scheme based on the one specified in the configuration file. Each RDD has five internal properties, namely, partition list, function to compute for every partition's dependency list on other RDDs, partitioner, and a list of locations to compute every partition. CHOPPER changes the partitioner properties to enable repartitioning across stages.

CHOPPER also supports dynamic updates to the Spark application configuration file based on the runtime information of current running workload. In particular, DAGScheduler periodically checks the updated configuration file and uses the updated partitioning scheme if available. This improves the partitioning efficiency and overall performance.

4.3.2 Determine Stage-Level Partition Scheme

The partition optimizer is responsible for computing a desirable partition scheme for each stage of a workload, given the collected workload history, and current input data and size. The optimizer not only considers the execution time and shuffle data size of a stage but also the shuffle dependencies between RDDs. In the following, we describe how this is achieved for the stage-level information. The use of global DAG information is discussed in Section 4.3.3.

Spark provides two types of partitioners, namely range partitioner and hash partitioner. Different data characteristics and data distributions require different partitioners to achieve optimal performance. Range partitioner creates data partitions with approximately same-sized ranges. RDD tuples that have keys in the same range are allocated to the same partition. Spark determines the ranges through sampling of the content of RDDs passed to it when creating a range partitioner. Thus, the effectiveness of a range partitioner highly depends on the data contents. A range partition scheme that distributes a RDD evenly is likely to partition another RDD into a highly-skewed distribution. In contrast, a hash partitioner allocates tuples using a hash function modulo of the number of partitions. The hash partitioner attempts to partition data evenly based on a hash function and is less sensitive to the data contents, and can produce even distributions. However, if the dataset has hot keys, a partition can become skewed in terms of load, as identical keys are mapped to the same partition. Consequently, the appropriate choice between the range partitioner or hash partitioner depends on the dataset characteristics and DAG execution patterns.

To compute the stage level partition scheme, we aim to minimize both the stage execution time and the amount of shuffle data. Considering stage execution time, and shuffle data that directly affects the execution time, enables us to capture the right task granularity. This prevents the partitions from both growing unexpectedly large—creating resource contention—or becoming trivially small—under-utilizing resources and incurring extra task scheduling overhead. The approach also implicitly alleviates task skew by filtering out inferior partition schemes.

Equations 4.1, 4.2, 4.3 and 4.4 describe the model learned and the objective function used to determine the optimal number of partitions. In particular, D denotes the size of input data for the stage, P denotes the number of partitions, t_{exe} represents the execution time of the stage, and $s_{shuffle}$ is the amount of shuffle data in the stage. t'_{exe} and $s'_{shuffle}$ denote the stage execution time and amount of shuffle data obtained using default parallelism, respectively. Given input data size, Equation 4.4 enables Chopper to determine the optimal number of partitions minimizing both execution time and the amount of shuffle data. By normalizing the execution time and the amount of shuffle data with respect to the respective values under default parallelism, we are able to capture both of our constraints into the same objective function. Constants α and β can be used to adjust the weights between the two factors. In our implementation, we set the constants to a default value of 0.5, making them equally important.

We model the execution time and the amount of shuffle data based on the input data size of the current stage and the number of partitions as shown in Equations 4.1 and 4.2. This is a coarse grained model, since it is independent of Spark execution details and focuses on capturing the relationship between input size, parallelism, execution time and shuffle data. In particular, we posit that the execution time increases with the input data size, obeying a combination of cube, square, linear, and sub-linear curves. The amount of shuffle data also increases or decreases with the number of partitions according to a combination of cube, square, linear, and sub-linear curves. Note that our model can capture most applications observed in the real-world use cases for Spark. However, it may not be able to model corner cases such as those with radically different behavior, e.g., workloads for which execution time grows with D^4. In general, we observe that such a model is simple and computationally efficient, yet powerful enough to capture applications with different characteristics via various coefficients of the model. When the cluster resources and other configuration parameters are fixed, the model fits the actual execution time and amount of shuffle data well with varying size of input data and number of partitions. The data points needed to train the models are gathered by the statistics collector. If the collected data points are not sufficient, Chopper can initiate a few test runs by varying the sampled input data size and the number of partitions and record the execution time and the amount of shuffle data produced. Chopper also remembers the statistics from the user workload execution in a production environment, which can be further leveraged to better train and model the current application behavior.

$$t_{exe} = a_1D^3 + b_1D^2 + c_1D + d_1D^{1/2} + e_1P^3 + f_1P^2 + g_1P + h_1P^{1/2}, \tag{4.1}$$

$$s_{shuffle} = a_2D^3 + b_2D^2 + c_2D + d_2D^{1/2} + e_2P^3 + f_2P^2 + g_2P + h_2P^{1/2}, \tag{4.2}$$

$$cost = \alpha t_{exe}/t'_{exe} + \beta s_{shuffle}/s'_{shuffle} \tag{4.3}$$

$$\min cost \tag{4.4}$$

Chopper trains two models using Equations 4.1 and 4.2 for every stage of a workload, one for range partitioning and the other for hash partitioning. Algorithm 1 presents how

Algorithm 1: Get Stage level Partition Scheme *getStagePar*.

```
Input: workload $w$, stage $s$, input size $d$
Output: ($partitioner$, $numPar$, $cost$)
begin
    $rModel$=getRangeParitionModel($w$,$s$)
    $hModel$=getHashPartitionModel($w$,$s$)
    ($numRangePar$,$rCost$)=getMinPar($rModel$,$d$)
    ($numHashPar$,$hCost$)=getMinPar($hModel$,$d$)
    if $rCost < hCost$ then
        return ($RangePartitioner$,$numRangePar$,$rCost$)
    else
        return ($HashPartitioner$,$numHashPar$,$hCost$)
```

Chopper calculates the optimized stage level partition scheme given workload w, stage s and input size d for the stage. The algorithm returns the partitioner, the optimal number of partitions used for stage s and the cost. Specifically, Chopper retrieves the trained models of stages for both range partitioning and hash partitioning from the workload database. After this, Algorithm 1 computes the optimal numbers of partitions with minimal cost for both range partitioning and hash partitioning using Equation 4.4. Finally, Chopper returns the partitioner that would incur the lowest cost along with the number of partitions to use.

4.3.3 Globally-Optimized Partition Scheme

After we compute the stage level partition scheme, a naive solution is to compute the optimal partition scheme for each stage independently and generate the Spark configuration file. This is shown in Algorithm 2. It gets the DAG information from workload database, iterates through the DAG, computes the desirable partition scheme for each stage, and adds to a list of partition scheme. Lastly, the algorithm returns the list of partition schemes, which is then used to generate the Spark configuration file for the current workload.

Although Algorithm 2 optimizes the partition scheme per stage, it misses the opportunities to reduce shuffle traffic because of the dependencies between stages and RDDs. For example, if stage-C joins the RDDs from stage-A and stage-B, the shuffle traffic introduced by join can be completely eliminated if the two use the same partition scheme and the joined partitions are allocated on the same machine. However, this cannot be achieved using Algorithm 2. If the computed optimal scheme of stage-A is $(Range, 100)$ and the optimal scheme of stage-B is $(hash, 200)$, the shuffle data cannot be eliminated, as the partition schemes of stage-A and stage-B are different. Even though stage-A and stage-B are optimally partitioned, the shuffle data of stage-C is sub-optimal. Since join and co-group operation are two of the most commonly used operations in Spark applications, poor partitioning will typically introduce significant shuffle overhead. Consequently, it is critical to optimize the join and co-group operation to decrease the amount of shuffle data as much as possible.

Another issue is that since the users are allowed to tune and specify customized partition scheme on their own, Chopper leaves the user optimization intact even when the computed optimal scheme disagrees with the user specified partition scheme. However, Chopper can choose to add an additional partition operation if the benefit of introducing the partition operation significantly outweighs the overhead incurred. For instance, consider a case where stage-B blows up the number of tasks to by a power of two from its previous stage-A (i.e., 100^2 tasks) due to the user-fixed partition scheme of stage-A. If Chopper coalesces the number of tasks of stage-A from 100 to 10, it would significantly reduce the number of tasks in stage-B from 10000 to 100.

Algorithm 2: Get workload partition scheme *getWorkloadPar*.

```
Input: workload $w$, DAG $dag$, input size $D$
Output: $parList$
begin
    if $dag == null$ then
        $dag$ = getDAG($w$)
    for $stage\ s\ in\ dag$ do
        $d$=getStageInput($w$, $s$, $D$)
        ($partitoner$, $numPar$, $cost$) = getStagePar($w$, $s$, $d$)
        $parList$.add($s$,$partitioner$,$numPar$,$cost$)
    return parList
```

To remedy this, Chopper determines the partition scheme by globally considering the entire DAG execution. As described in Algorithm 3, Chopper first groups the DAG graph based on the stage dependencies. The grouping of DAG graph is started from the end stages of the graph and iterated towards the source stages. The grouping is based on the join operations or partition dependencies. The stages with join operations are grouped into a subgraph. The partition dependencies refer to the cases where the number of stages is determined by the previous stage, thus Chopper cannot change the partition scheme. After the DAG of the workload is regrouped, the node within the new DAG can either be a stage or a subgraph that consists of multiple stages. If the node is a stage, the optimal partition scheme is computed using Algorithm 1. Otherwise, the node is a subgraph, where the optimal partition scheme is computed differently. Specifically, we iterate through all the nodes within the subgraph and get the optimal partition scheme for each node, we then compute the cost of applying each partition scheme to all the applicable nodes in the graph and return the partition scheme that has the minimal cost.

Finally, after we compute the globally optimal partition scheme, we check whether the stage partition is allowed to be changed. If not, and the partition scheme is different, we then check whether it is beneficial to insert a new repartitioning phase by comparing the cost using original partitioning to the cost of the new repartitioning phase together with the cost of optimized partition scheme. If the benefit outweighs the cost by a factor of γ, we choose to insert a new partition phase into the DAG graph. We empirically set γ to 1.5 to tolerate the model estimation error.

Algorithm 3: Get globally optimized partition scheme.

```
Input: workload $w$, input size $D$
Output: $parList$
begin
    $dag$=getReGroupedDAG($w$)
    for $node\ s\ in\ dag$ do
        $d$=getStageInput($w$, $s$, $D$)
        if $s\ isInstanceOf\ Stage$ then
            ($partitoner$, $numPar$, $cost$) = getStagePar($w$, $s$, $d$)
        else
            ($partitoner$, $numPar$, $cost$) = getSubGraphPar($w$, $s$, $d$)
        if $s\ isFixed$ then
            $curCost$ = getCost($w$, $s$, getPartitioner($w$, $s$), getNumPar($w$, $s$))
            $optCost$ = $cost$ + getRepartitionCost($w$, $s$, $partioner$, $numPar$)
            if $optCost < curCost$ then
                $s' = s +$ "$repartitionstage$"
                $parList$.add($s'$, $partitioner$, $numPar$, $optCost$)
        else
            $parList$.add($s$, $partitioner$, $numPar$, $cost$)
    return parList
Function $getSubGraphPar$
    Input: workload $w$, DAG $dag$, input size $D$
    Output: $paritioner, numPar, cost$
    $parList$ = getWorkloadPar($w$, $dag$, $D$)
    $min = parList(0)$
    for $s\ in\ parList$ do
        $cost$ = getCost($w$, $dag$, $s.partitioner$, $s.numPar$)
        if $cost < min.cost$ then
            $min = s$
    return ($min.partitioiner, min.numPar, min.cost$)
Function $getCost$
    Input: workload $w$,DAG $dag$, $partitioner$, $numPar$
    Output: $cost$
    for $stage\ s\ in\ dag$ do
        if $partitioner == range$ then
            $rModel$ = getRangePartitionModel($w$, $s$)
            $cost$ += Equation 4.3
        else
            $hModel$ = getHashParitionModel($w$, $s$)
            $cost$ += Equation 4.3
    return $cost$
```

4.4 Evaluation

In this section, we evaluate Chopper and demonstrate its effectiveness on the cluster described earlier in Section 7.2. We use three representative workloads from SparkBench: KMeans, PCA, and SQL. KMeans [83] is a popular clustering algorithm that partitions and

Workload	KMeans	PCA	SQL
Input Size (GB)	21.8	27.6	34.5

Table 4.1: Workloads and input data sizes.

clusters n data points into k clusters in which each data point is assigned to the nearest center point. The computation requirement of this workload change according to the number of clusters, the number of data points, and the machine learning workload that exhibits different resource utilization demand for different stages during the process of iteratively calculating k clusters. PCA [95] is a commonly used technique to reduce the number of features in various data mining algorithms such as SVM [63] and logistic regression [87]. It is both computation and network-intensive machine learning workload that involves multiple iterations to compute a linearly uncorrelated set of vectors from a set of possibly correlated ones. SQL is a workload that performs typical query operations that count, aggregate, and join the data sets. Thus, SQL represents a common real world scenario. SQL is compute intensive for count and aggregation operations and shuffle intensive in the join phase. The input data is generated by the corresponding data generator within SparkBench. The input data size for each workload is shown in Table 4.1. The experiments for vanilla Spark are conducted with the default configuration, which is set to 300 partitions for all the workloads. We run all of our experiments three times, and the numbers reported here are from the average of these runs. Moreoer, we clear the OS cache between runs to preclude the impact of such caching on observed times.

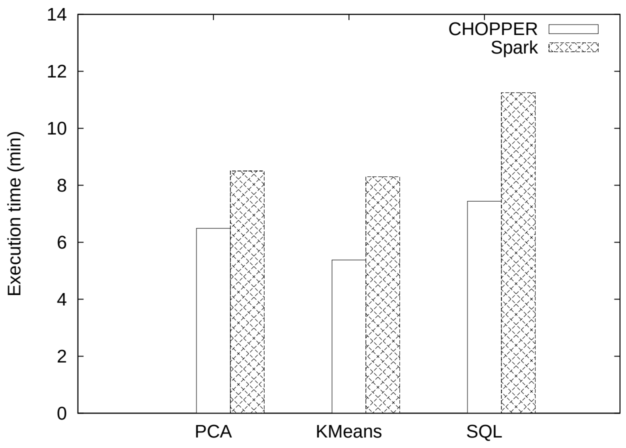


Figure 4.7: Execution time of Spark and CHOPPER.

4.4.1 Overall Performance of Chopper

Our first test evaluates the overall performance impact of CHOPPER. Fig. 4.7 illustrates the total execution time of three workloads comparing CHOPPER against standard vanilla

Spark. The reported execution time includes the overhead of repartitioning introduced by Chopper. We can see that Chopper achieves overall improvement in the execution time by 23.6%, 35.2% and 33.9% for PCA, KMeans and SQL, respectively. This is because Chopper effectively detects optimal partitioner and the number of partitions for all stages within each workload. Chopper also performs global optimization to further reduce network traffic by intelligently co-partitioning dependent RDDs and inserting repartition operations when the benefits outweigh the cost. We also observe that Chopper is effective for all types of workloads that exhibit different resource utilization characteristics. The repartition method of Chopper implicitly reduces the potential data skew and determines the right task granularity for each workload, thus improving the cluster resource utilization and workload performance. Thus, Chopper shows significant reduction in the overall execution time for all the three workloads. The model training of Chopper is conducted offline, and thus is not in the critical path of workload execution. Moreover, the overhead of repartitioning is negligible as it involves solving a simple linear programming problem.

4.4.2 Timing Breakdown of Execution Stages

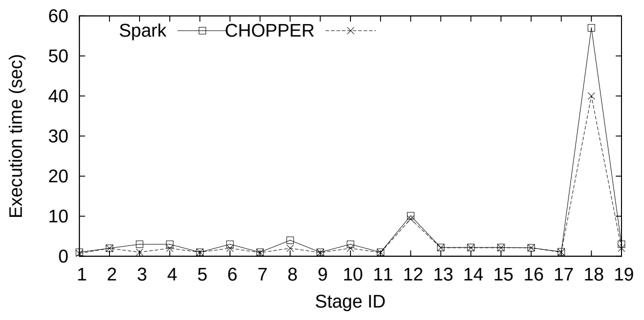


Figure 4.8: Execution time per stage breakdown of KMeans.

	Chopper	**Spark**
Execution Time (sec)	250	372

Table 4.2: Execution time for stage 0 in KMeans.

To better understand how dynamic partitioning of Chopper helps to improve overall performance, in our next test, we examine the detailed timing breakdown of individual workload stages. Fig. 4.8 depicts the execution time per stage for KMeans. We show the execution time of stage-0 separately in Table 4.2 since the execution time of stage-0 and that of other stages differs significantly. We see that Chopper reduces execution time of each stage for KMeans compared to vanilla Spark, as Chopper is able to customize partition schemes

StageID	0	1	2	3	4	5	6	7	8	9	10	11	12 - 17	18	19
Chopper	210	210	300	720	300	720	300	720	300	720	300	720	210	380	210
Spark	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300

Table 4.3: Repartition of stages using Chopper.

for each stage according to associated history and runtime characteristics. Table 4.3 shows the number of partitions used by Chopper for different stages compared to vanilla Spark. Stages 12 to 17 are iterative, and thus are assigned the same number of partitions. We see that Chopper effectively detects and changes to the correct number of partitions for this workload rather than using a fixed (default) value throughout the execution.

4.4.3 Impact on Shuffle Stages

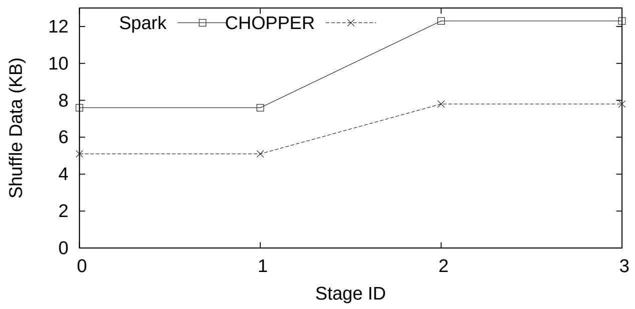


Figure 4.9: Shuffle data per stage for SQL.

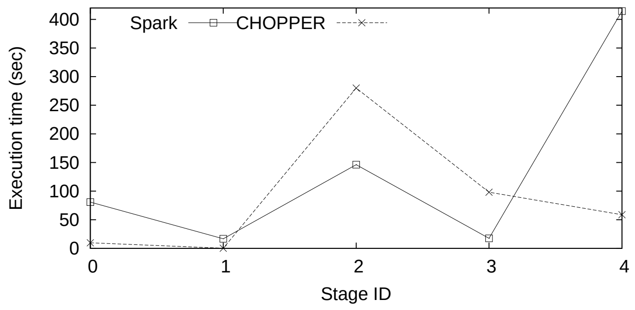


Figure 4.10: Execution time per stage breakdown of SQL.

In our next test, we use a shuffle-intensive workload, SQL, to study how Chopper reduces the shuffle traffic by automatically recognizing and co-partitioning dependent RDDs. Fig. 4.9 shows that the shuffle data for all four stages is less under Chopper compared to vanilla Spark. Stage-4 (not shown in the figure) has the same amount of shuffle data for SQL workload using Chopper or Spark (i.e., 4.7 *GB*). However, as seen in Fig. 4.10, stage-4 takes comparatively shorter time to execute using Chopper versus Spark. This is because, stage 4 is divided into 4 sub-stages where the first two sub-stages have a shuffle write data of 1.9 GB and 2.8 GB. Chopper combines these two sub-stages for shuffle write and provides the third sub-stage for the shuffle data to be read. This greatly reduces the execution time for stage 4 as seen in Fig. 4.10. Thus, we demonstrate that Chopper can effectively detect dependent RDDs and co-partition them to reduce the shuffle traffic and improve the overall workload performance.

4.4.4 Impact on System Utilization

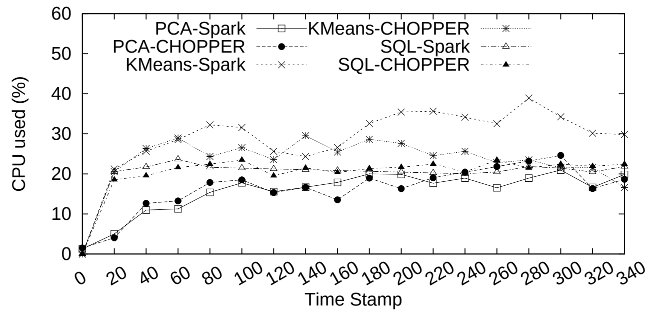


Figure 4.11: CPU utilization.

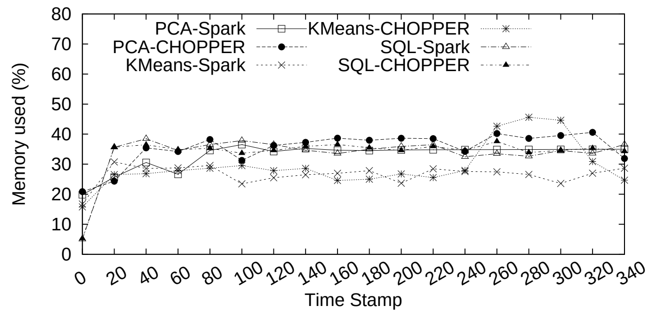


Figure 4.12: Memory utilization.

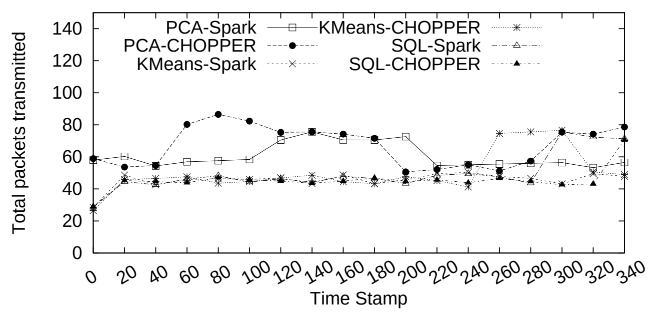


Figure 4.13: Total transmitted and received packets per second.

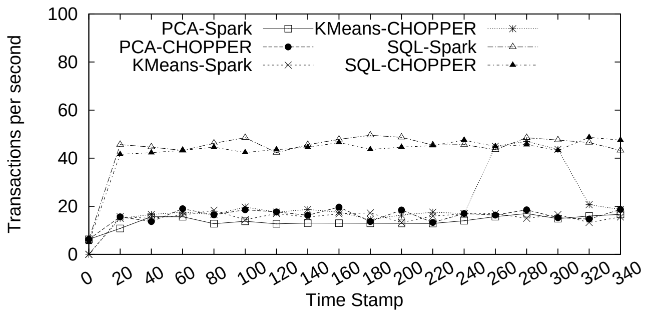


Figure 4.14: Transactions per second.

In our next experiment, we investigate how Chopper impacts the resource utilization of all the studied workloads. Fig. 4.11, 4.12, 4.13 and 4.14 depict the CPU utilization, memory utilization, total number of transmitted and received packets per second, and the total number of read and write transactions per second, respectively, during the execution of the workloads under Chopper and vanilla Spark. The numbers show the average of the statistics collected from the six nodes in our cluster setup. We observe that the performance of Chopper is either equivalent or in most of the cases better than the performance of vanilla Spark for the studied workloads. In some cases, Chopper shows improved transactions per seconds as compared to vanilla Spark because of the high throughput and improved system performance.

These experiments show that the performance (computed on the basis of execution time and shuffle data) improves under Chopper compared to vanilla Spark. Also, these improvements in Chopper yield comparable or better system utilization compared to vanilla Spark.

4.5 Chapter Summary

In this chapter, we design Chopper, a dynamic partitioning approach for in-memory data analytic platforms. Chopper determines the optimal number of partitions and the partitioner for each stage of a running workload with the goal of minimizing the stage execution time and shuffle traffic. Chopper also considers the dependencies between stages, including join and co-group operations, to further reduce shuffle traffic. By minimizing the stage execution time and shuffle traffic, Chopper implicitly alleviates the task data skew using different partitioners and improves the task resource utilization through optimal number of partitions. Experimental results demonstrate that Chopper effectively improves overall performance by up to 35.2% for representative workloads compared to standard vanilla Spark.

Our current implementation of Chopper has to re-train its models whenever the available resources are changed. In future, we plan to explore the per-stage performance models that can work across different resource configurations, i.e., clusters. We will also explore how

Chopper behaves under failures. These will further improve the applicability of Chopper in a cloud environment, where compute resources are failure-prone and scaled as needed.

Chapter 5

File System Monitoring for Large Scale Storage Systems

5.1 Introduction

Big data science relies on sophisticated workflows that need large storage systems to meet the demands of their data-intensive workloads. According to a recent report from National Energy Research Scientific Computing Center (NERSC) [14], the total volume of data stored at NERSC has grown at an annual rate of 30 percent over the past 10 years. Data may be stored for short to long periods, be accessible to dynamic groups of collaborators, and be operated upon by various actors. As data generation volumes and velocities continue to increase, the rate at which files are created, modified, deleted, and acted upon (e.g., permission changes) make manual oversight and management infeasible.

While research automation [21, 56] offers a potential solution to these problems it is first necessary to be able to capture events in real-time and at scale across a range of large-scale storage systems. Such events may then be used to automate the data lifecycle (performing backups, purging stale data, etc.), report usage and enforce restrictions, enable programmatic management, and even autonomously manage the health of the system. Enabling scalable, reliable, and standardized event detection and reporting will also be of value to a range of infrastructures and tools, such as software-defined cyberinfrastructure (SDCI) [75], auditing [41], and automating analytical pipelines [56, 57]. Such systems enable automation by allowing programs to respond to file events and initiate tasks.

Many small-scale local storage systems provide mechanisms to detect and report data events, such as file creation, modification, and deletion. Tools such as inotify [124], kqueue [108], and FileSystemWatcher [131] enable developers and applications to respond to data events. However, such tools are not scalable and suitable for large-scale parallel file systems, like, Lustre [12] and IBM's Spectrum Scale [168] (formally known as GPFS), and object-based storage systems, like Ceph [201] and Gluster [79]. Such large-scale storage systems often maintain an internal metadata collection catalog, such as Lustre's *Changelog*, IBM Spectrum Scale's *mmaudit*, Ceph's *Journal*, and Gluster's *libgfchangelog*, which enables developers to query data events. However, each of these tools provides a unique API and event description language. For example, a file creation action may be recorded as a *01CREAT* event in Lustre's Changelog, and an *openc* event in Ceph's Journal. Furthermore, the difference in the architecture of parallel file systems and object-based storage systems makes developing a generalized file system event monitoring tool for large-scale storage systems non-trivial.

In this chapter, we present a generalized, and scalable, storage system monitor, called `FSMonitor`, for Storage Monitor. `FSMonitor` provides a common API and event representation for detecting and managing data events from different large-scale storage systems (parallel file systems and object-based storage systems). We have architected `FSMonitor` with a modular Data Storage Interface (DSI) layer to plug-in and use custom monitoring tools and services. We leverage the internal metadata collection catalog found in such storage systems to develop a scalable monitor architecture that is capable of detecting, resolving, and reporting these events. We choose Lustre as our implementation platform for parallel file systems because Lustre is used by 60% of the top 500 supercomputers [72]. For our implementation of `FSMonitor` on object-based storage system, we select Ceph as it is one of the most popular and open-source object stores, with companies such as Bloomberg, Cisco, and Deutsche-Telekom using Ceph as their storage backend [5]. While our scalable monitor has so far only been applied to Lustre and Ceph storage systems, its design makes it applicable to other large-scale storage systems with metadata catalogs, such as IBM's Spectrum Scale and Gluster.

We evaluate `FSMonitor` on real-world deployments of both the Lustre file system and the Ceph object store, and demonstrate its capabilities by using it to drive a research automation system. Our experiments show that `FSMonitor` scales well on a 897 TB Lustre store, and a 8 TB Ceph store, where it is able to process and report 37 948 events per second on Lustre, more than 2000 events per second on Ceph, while providing a standard event representation on both storage systems. We also implement and evaluate a use case for `FSMonitor`, in the form of a file system indexer that uses `FSMonitor` to improve the performance and efficiency of metadata re-indexing.

5.2 Background

In this section, we describe parallel file systems and object-based storage. We focus specifically on the Lustre and Ceph storage systems supported by `FSMonitor`. We describe how the Lustre Changelog metadata catalog and the Ceph metadata Journal can be used to detect and report events.

5.2.1 Parallel File System

A parallel file system is designed to stripe data blocks in parallel, across multiple storage devices on multiple servers. Multiple clients can access a file system simultaneously and can perform I/O operations on the various stripes of a file in parallel. Typically, it consists of *clients* that read or write data to the file system, *data servers* where data is stored, *metadata servers* that manage the metadata and placement of data on the data servers, and networks to connect these components. This level of parallelism is transparent to the clients, for whom it seems as though they are accessing a local file system. Therefore, important functions of a parallel file system include avoiding potential conflict among multiple clients, and ensuring data integrity and system redundancy.

Monitoring a parallel file system is challenging as events may be generated across various components in the system and they then pass through one of several metadata servers. Generally, the larger the data store, the more metadata servers required to manage the cluster. Further, as a parallel file system aims to deliver transparent parallelism, it is crucial that even the event monitoring mechanisms appear to users as if they are the same as a local file system.

Lustre Parallel File System

As seen in the Chapter 3, Lustre is designed for high performance, scalability, and high availability. It employs a client-server network architecture. A Lustre deployment is comprised of one or more Object Storage Servers (OSSs) that store file contents; one or more Metadata Servers (MDSs) that provide metadata services for the file system and manage the Metadata Target (MDT) that stores the file metadata; and a single Management Server (MGS) that manages the system.

The *Management Server* (MGS) is responsible for storing the configuration information for the entire Lustre file system. This persistent information is stored on the *Management Target* (MGT). *Metadata Servers* (MDS) manages namespace operations and are responsible for one *Metadata Target* (MDT). Namespace metadata, such as directories, filenames, file layouts, and access permissions are stored in the associated MDT. Every Lustre file system must have at least one MDT. A large file system may require more than one MDT to store its metadata, and therefore, Lustre versions after 2.4 support a distributed namespace. A distributed namespace means metadata can be spread across multiple MDTs. MDS_0 and MDT_0 act as the root of the namespace, and all other MDTs and MDSs act as their children.

Lustre provides a *Changelog* to query the metadata stored in the MDT. Developers can create a Changelog listener and subscribe to a specific MDT, allowing them to retrieve and purge metadata records. `FSMonitor` uses the Lustre Changelog to keep track of file system events.

Object Storage Servers (OSS) store file contents. Each file is stored on one or more *Object Storage Targets* (OST) mounted on an OSS. Applications access file system data via Lustre clients which interact with OSSs directly for parallel file accesses. The internal high-speed data networking protocol for Lustre file system is abstracted and is managed by the Lustre Network layer.

5.2.2 Object Storage

Object storage [69] runs on the principle of assigning a unique identifier and metadata to each piece of data to be stored in the system. These data units are accessed as objects rather than a stream of bytes. Figure 5.1 shows how object-based storage model differentiates from the traditional block-based model.

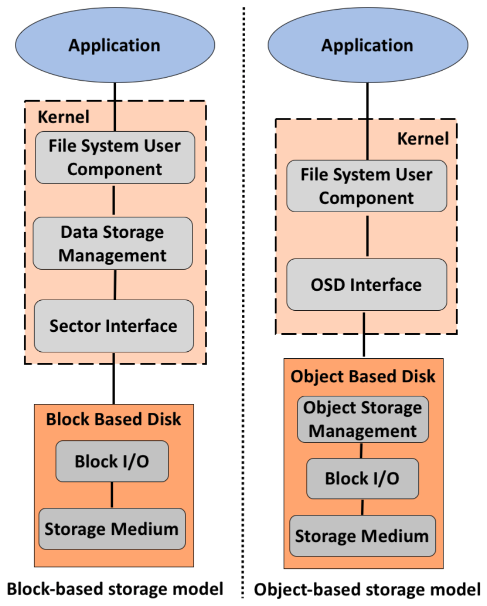


Figure 5.1: Comparison of block-based and object-based storage models.

In the traditional block-based storage model, the host operating system is responsible for translating the data layout and application requests into logical block addresses. In the object-based model, the responsibility of translation into block addresses is moved to the object-based disk.

Therefore, data retrieval is much faster in the object-based storage model by forgoing the need to traverse through the tiered architecture of traditional file systems and instead, storing all objects in a common storage. Typically, the metadata and data are segregated into different logical pools to facilitate more efficient metadata driven operations on the system. Object-based models are able to achieve high scalability and are appropriate in areas requiring high data throughput, such as Internet-of-Things (IoT), and streaming services like Netflix [15]. We implement a `FSMonitor` interface for the Ceph object store, which is explained below.

Ceph Object Storage

Ceph [201] storage is a popular object stores [4, 17]. Figure 5.2 shows its architecture.

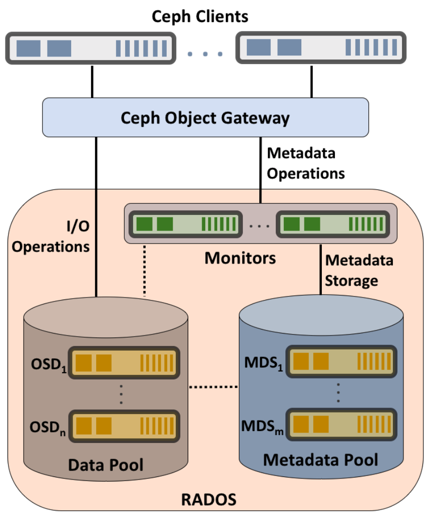


Figure 5.2: An overview of Ceph architecture.

The primary component of Ceph's infrastructure is the distributed object store - *RADOS* (Reliable Autonomic Distributed Object Store). Data are stored as objects in logical groups called pools. The metadata pool is separated from the data pools to aid faster execution of metadata level operations, such as list and rename. The pools host logical namespaces known as Placement Groups (PGs), which group objects into fragments within a pool. Each placement group is mapped to a set of *Object Storage Daemons* (OSDs) where the final placement of incoming data objects is done.

The *Ceph client* interacts with RADOS to store data by providing the object name and the pool name using the *Ceph object gateway*. The interaction between the Ceph client and the OSDs is coordinated by *Metadata Servers* (MDSs), placed in the metadata pool. Each MDS maintains a *Journal* to assimilate the series of metadata level manipulations made to the storage system. `FSMonitor` uses the MDS Journal to collect file system events from Ceph. All metadata operations go through the Ceph *Monitor*, which is used to maintain a cluster map, that is consulted by the Ceph client. The cluster map is a collection of the Monitor map, OSD map, PG map, and MDS map. To ensure that Ceph does not succumb to a single point of failure, Ceph tries to attain a decentralized architecture by allowing multiple MDSs and Monitors in the cluster. The implementation of `FSMonitor` is done atop the Ceph File System (CephFS), which is built on top of RADOS. It is POSIX compliant with a traditional file system interface.

5.2.3 Monitoring Storage System Events

Most storage systems maintain a catalog for recording file system events. For example, Lustre maintains a *Changelog*, Ceph a *Journal*, IBM Spectrum Scale [168] a file audit log (*mmaudit*), and GlusterFS [79] a *libgfchangelog* to track metadata events. As `FSMonitor` is implemented atop Lustre and Ceph file systems, we explain Lustre Changelog and Ceph Journal below. Lustre and Ceph are representative of other parallel file systems and object stores, and thus `FSMonitor` can be extended to build a monitoring solution for other large-scale storage systems as well.

Lustre Changelog

ID	Type	Time	Date	Flags	Target FID	Parent FID	Target
11	01CREAT	21:18:47.30	2020.06.20	0x0	t=[0x5716:0x626c:0x0]	p=[0x5716:0xe7:0x0]	hello.txt
12	17MTIME	21:18:47.32	2020.06.20	0x7	t=[0x5716:0x626c:0x0]		hello.txt
13	08RENME	21:18:47.41	2020.06.20	0x1	t=[0x5716:0x17a:0x0]	p=[0x5716:0xe7:0x0] s=[0x5716:0x626b:0x0] sp=[0x5716:0x626c:0x0]	hello.txt hi.txt
14	02MKDIR	21:18:47.42	2020.06.20	0x0	t=[0x5716:0x626d:0x0]	p=[0x5716:0xe7:0x0]	okdir
15	06UNLNK	21:18:47.43	2020.06.20	0x0	t=[0x5716:0x626b:0x0]	p=[0x5716:0xe7:0x0]	hi.txt

Table 5.1: A sample Lustre ChangeLog record showing *Create File*, *Modify*, *Rename*, *Create Directory*, and *Delete File* events.

Table 5.1 shows sample records in Lustre's Changelog. We run a script and see the events recorded in the Changelog. The script first creates and then modified a file, *hello.txt* and then renames the file to *hi.txt*. A directory named *okdir* is then created. Finally, it deletes the file.

Each tuple in Table 5.1 represents a file system event, specified via a *EventID* (record number for the Changelog entry), *Type* (file system event type), *Timestamp, Datestamp* (date and time of the event occurrence), *Flags* (masking for the event), *Target FID* (file identifier of the target file/directory on which the event occurred), *Parent FID* (file identifier of the parent directory of the target file/directory), and *Target Name* (the file/directory name that triggered the event). It is evident that the Parent and Target FIDs need to be resolved to their original names before they can be sent to the client. The following event types are recorded in the Changelog:

- `CREAT`: Creation of a regular file.
- `MKDIR`: Creation of a directory.
- `HLINK`: Hard link.
- `SLINK`: Soft link.

- `MKNOD`: Creation of a device file.
- `MTIME`: Modification of a regular file.
- `UNLNK`: Deletion of a regular file.
- `RMDIR`: Deletion of a directory.
- `RENME`: Rename a file or directory from.
- `RNMTO`: Rename a file or directory to.
- `IOCTL`: Input-output control on a file or directory.
- `TRUNC`: Truncate a regular file.
- `SATTR`: Attribute change.
- `XATTR`: Extended attribute change.

Note in Table 5.1 that Target FIDs are enclosed within $t = []$, and parent FIDs within $p = []$. *MTIME* event does not have a parent FID. *RENME* event has additional FIDs, $s = []$ denoting a new file identifier to which the file has been renamed, and $sp = []$ gives the file identifier for the original file. These features are important when resolving FIDs.

Ceph Journal

EventID	Type	ctime	dentry	Root_dentry	dirHash
0x8a6422d8	openc	2020-06-20 22:44:25.425743	full bits = hello.txt	/mnt/mycephfs	0
0x8a6428e7	cap update	2020-06-20 22:44:28.159760	full bits = hello.txt	/mnt/mycephfs	0
0x8a642efb	rename	2020-06-20 22:44:51.226914	full bits = hi.txt null bits = hello.txt	/mnt/mycephfs	0
0x8a643534	mkdir	2020-06-20 22:44:59.111970	full bits = okdir	/mnt/mycephfs	2
0x8a643c5d	unlink_local	2020-06-20 22:45:07.665030	full bits = stray3 null bits = hi.txt	/mnt/mycephfs	0

Table 5.2: A sample Ceph Journal record showing *Create File*, *Modify*, *Rename*, *Create Directory*, and *Delete File* events.

Table 5.2 shows the sample records saved in a Ceph Journal when we run the same script as for generating the Lustre Changelog, shown in Table 5.1. Each row in Table 5.2 represents an event in the object store. Every row consists of an *EventID* (journal entry number), *Type* (type of file system event), *ctime* (date and timestamp when the event occurred), *dentry* (file or directory name on which the event occurred), *Root_dentry* (parent directory name for the dentry file or directory), and *dirHash* (an identifier to check if the *dentry* is a directory). In file systems, an inode (index node) contains information about a file. A dentry (directory entry) is used to keep track of the hierarchy of files in directories. Each dentry maps an inode number to a file name and a parent directory. For file system events where an inode entry is removed from the system, *dentry* has entries for *full bits* and *null bits*. For example, when *hello.txt* is renamed to *hi.txt*, the dentry value for *full bits* is *hi.txt*, and dentry value for *null bits* is *hello.txt*. Similarly, when file *hi.txt* is deleted, *null bits* value in dentry is *hi.txt*. The value for *dirHash* is non-zero when *full bits* of dentry points to a directory.

It should be noted from Tables 5.1 and 5.2 that there is no standard representation for a file system event in parallel file systems and object stores. A file creation event in Lustre is recorded as `01CREAT`, while in Ceph, it is recorded as `openc`. Similarly, a file modification event is `17MTIME` in Lustre and `cap update` in Ceph; a file or directory rename event is `08RENME` in Lustre and `rename` in Ceph; a directory creation event is `02MKDIR` in Lustre and `mkdir` in Ceph; and a file deletion event is `06UNLNK` in Lustre and `unlink_local` in Ceph. Thus a scalable monitor is needed that can create a standard representation of file system events.

5.2.4 Prior Work

Monitoring file system events in large scale storage systems needs specialized tools. For object storage systems, there is no such tool which is widely used. Oral et al. [143] developed an efficient object storage journaling tool for a distributed parallel file system. They however focused on a hardware solution by using external journals on solid state devices. `FSMonitor` focuses on providing a generic software solution that can work on all storage infrastructure. Another monitoring tool [53] was developed for the ATLAS project [1] which uses a scalable publisher-subscriber model to record metadata events. This work is however specific to the ATLAS project. `FSMonitor` uses the pub-sub model from this work to make the monitoring solution scalable as well as generic.

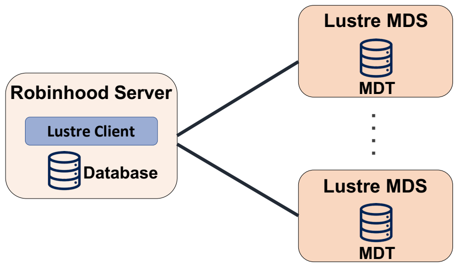


Figure 5.3: Robinhood architecture.

The Lustre file system has some specialized tools [133] for metadata event monitoring, of which Robinhood [107] is the most widely used. The Robinhood Policy Engine is capable of collecting events from Lustre file systems and using them to drive a policy engine to automate data management tasks, such as purging old files. Its architecture is shown in Figure 5.3. Robinhood uses an iterative approach to collect event data from metadata servers. A Robinhood server runs on the Lustre client and queries each MDS for events by querying the Changelogs. It then saves the events in a database on the Lustre client. For multiple MDSs, Robinhood polls the MDSs sequentially, in a round robin fashion. In contrast, `FSMonitor` employs a high performance message queue to concurrently collect, report, and aggregate events from all MDSs. This approach allows `FSMonitor` to far exceed Robinhood event collection performance, as we will show in Section 5.4.3.

Summary: There is a need for a general event monitoring solution that can be applied to various storage systems including both parallel file systems (such as Lustre) as well as object-based storage systems (such as Ceph). The monitor needs to have a standard event representation. In `FSMonitor`, we standardize all event representations to the *inotify* [115] format from Linux, as this is the most widely used representation in industry [10]. The monitor also needs to be scalable so as to be able to capture all the events in a large-scale storage system. We implement `FSMonitor` for the Lustre file system and Ceph object store.

5.3 `FSMonitor` System Design

Existing monitoring solutions for local file systems (Linux, macOS) cannot be extended to distributed environments and existing solutions for parallel file systems cannot be generally applied to different storage systems. `FSMonitor` provides a standard event detection interface across all large scale storage systems, and extends the reliability of the underlying event detection system by providing increased resiliency. `FSMonitor` is designed to be applied to arbitrary large scale storage systems through a modular data storage interface which can be implemented to connect to arbitrary event interfaces (parallel file system and object stores). It provides a standardized API to collect and process file system events, and is a scalable and high-performance system that is capable of detecting many thousands of events per second.

5.3.1 Logical View of `FSMonitor`

Figure 5.4 illustrates the three-layer architecture of `FSMonitor`. The *Data Storage Interface* (DSI) layer provides an abstraction for interfacing with different storage systems. The *resolution* layer is used to process events and standardize their format, and the *interface* layer enables client users and programs to communicate with `FSMonitor`. Here we describe each of these layers in detail, explaining their design and capabilities.

Data Storage Interface Layer

The lowest level of `FSMonitor` is responsible for interfacing with the underlying storage system to capture events and report them to the resolution layer for processing. We employ

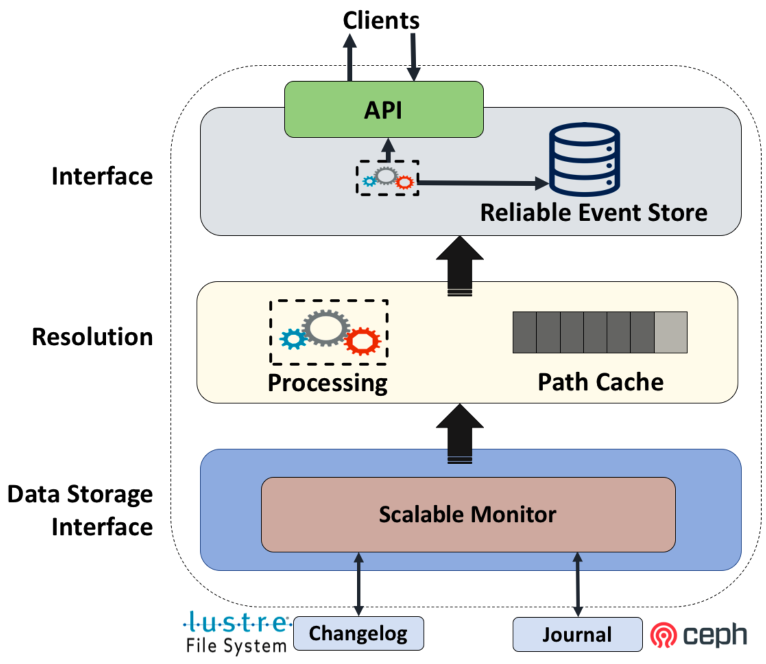


Figure 5.4: `FSMonitor` architecture, showing the interface, resolution, and DSI layers.

a modular architecture via which arbitrary monitoring interfaces can be integrated into `FSMonitor`. We provide a set of DSIs to interface with parallel file systems and object storage systems. Our DSI layer is responsible for selecting the appropriate catalog to monitor for the given storage device (Changelog in Lustre, and Journal in Ceph). After the events have been collected, they are propagated to the resolution layer.

Resolution Layer

The resolution layer is responsible for reliably recording and aggregating events from the DSIs and then reporting these events to the interface layer. This layer includes a queue to receive and manage events until they are processed. As events are received from a DSI plug-in, they are immediately placed in the processing queue. The events are then processed to resolve and dereference paths such that events can be transformed into various representations. This step is explained in Section 5.3.2. Rather than defining yet another event representation, we instead transform all events into the widely used *inotify* event representation format. Thus, `FSMonitor` provides a generalized event representation for all large scale storage systems. The resolution layer also provides optimizations that improve the overall event processing performance, such as caching and batching capabilities.

Interface

The topmost layer provides an interface for users and programs to interact with `FSMonitor`. This layer is responsible for reporting events and replying to requests. Programs can use the interface to capture events as they happen. The interface layer also provides batching capabilities to report events in groups. If an event identifier is provided, `FSMonitor` only reports events that have occurred since that event. To provide fault tolerance, this layer also stores all events received from the *resolution* layer into an event store (database). In the event of failures, after client recovery, once events have been retrieved from `FSMonitor`, they are flagged as having been reported and can be removed from the database. The size of this database is configurable and can be adjusted depending on the resources available to `FSMonitor`.

5.3.2 Storage System View of `FSMonitor`

Figure 5.5 illustrates a deployment of `FSMonitor` on large scale storage systems. Our monitor uses a publish-subscribe model, where the metadata is acquired from each MDS's metadata catalog via a *collector*, processed and cached, then published to the *aggregator*. The aggregator gathers events from each collector for later consumption. The publish-subscribe model provides for scalable event collection, as proven for monitoring Lustre server statistics [150, 152, 196]. The aggregator, and collector components are part of the architecture of `FSMonitor`.

Therefore, mapping the logical view of `FSMonitor` with its storage system view, we have the DSI layer on collectors, resolution layer comprises of both collectors and aggregator, and the interface layer spanning part of aggregator and the consumers. For parallel file systems like Lustre, the collector service is placed on the MDSs, the aggregator service on the MGS, and the consumers are the file system clients. This is also similar to object storage systems, like Ceph, where the collector service is on the MDSs in the metadata pool, aggregator service is on the monitor, and the consumers consist of the object storage clients. We divide the overall storage system design of `FSMonitor` into four steps: *Detection*, *Processing*, *Aggregation*, and *Consumption*. We describe each step below.

Detection: Each collector service is responsible for extracting file system events from one MDS metadata catalog. Deploying collectors on individual MDSs enables every MDS to be monitored in parallel. The collector is responsible for detecting the type of storage system: either parallel file system or object-based storage. The collector monitors the catalog (Changelog in Lustre, and Journal in Ceph) and for every new event that is extracted from the local metadata catalog, the event needs to be processed before it can be published to the aggregator.

Processing (Lustre): We explain first the processing step for Lustre. As seen in Section 6.2.4, every event in the Lustre Changelog is associated with either a target or a parent

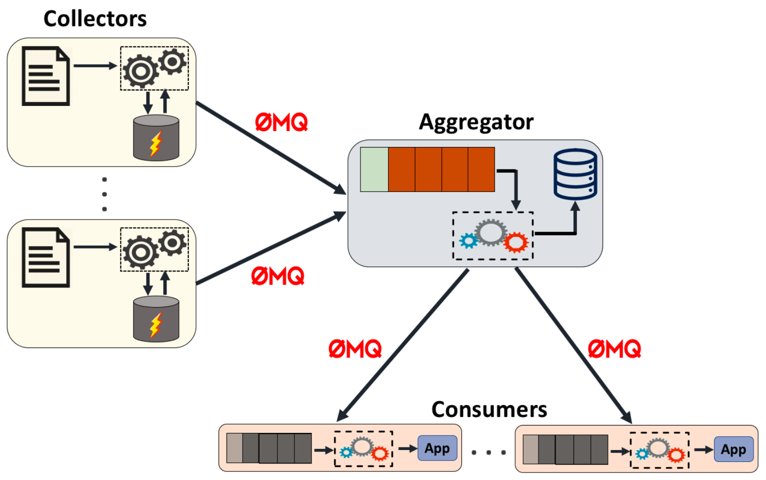


Figure 5.5: Infrastructural view of `FSMonitor`.
On Lustre: Collectors=MDSs, Aggregator=MGS, Consumers=Lustre Clients.
On Ceph: Collectors=MDSs, Aggregator=Monitor, Consumers=Ceph Clients.

file identifier (FID), or both. However, these FIDs are not necessarily interpretable by external applications, and thus must be processed and resolved to absolute path names.

The Lustre *fid2path* tool resolves FIDs to absolute path names. Whenever a new file system event is detected by a collector, we use this tool to convert raw event tuples into application-friendly file paths when the event is being published. However, the *fid2path* tool is slow and can delay the reporting of events. For example, in Section 5.4.3 we show that this delay can cause a decrease of 14.9% in the event reporting rate. To minimize this overhead, we implement a Least Recently Used (LRU) Cache in the aggregator to store mappings of FIDs to source paths.

Algorithm 7 shows the aggregator's processing steps for a Lustre Changelog. Events are processed in batches. The cache is used to resolve FIDs to absolute paths. Whenever an entry is not found in the cache, we invoke the *fid2path* tool to resolve the FID and then store the mapping *(fid – path)* into the LRU cache. In the case of `UNLNK` and `RMDIR` events, resolving target FIDs will give an error because that FID has already been deleted by the file system. Therefore, parent FIDs need to be resolved. If resolving parent FID raises an error, it means the parent directory has also been deleted. `FSMonitor` resolves this and gives the event as 'ParentDirectoryRemoved'. As seen in Section 6.2.4, `RENME` events are provided with the old path ($sp = []$) and new path FIDs ($s = []$). Therefore, for `RENME` events, instead of target FID, old path and new path FIDs need to be resolved. Finally, events are published to the aggregator.

After processing a batch of file system events from the Lustre Changelog, the collector will purge the Changelogs. A pointer is maintained to the most recently processed event tuple and all previous events are cleared from the Changelog. This helps reduce the overburdening of the Changelog with stale events.

Algorithm 4: Processing Lustre Changelog events.

```
Input: Lustre path lpath, Cache cache, MDT ID mdt
Output: EventList
while true do
    events = read events from mdt Changelog
    for event e in events do
        resolvedEvent = processEvent(e)
        EventList.add(resolvedEvent)
    Clear Changelog in mdt
    return (EventList)
Function processEvent
    Input: Event e
    Output: resolvedEvent
    Extract event_type, time, date from e
    try:
        path = cache.get(targetFID)
        if targetFID not found in cache then
            path = fid2path(targetFID)
            cache.set(targetFID, path)
    catch fid2pathError:
        try:
            if event_type is UNLNK or RMDIR then
                path = cache.get(parentFID)
                if parentFID not found in cache then
                    path = fid2path(parentFID)
                    cache.set(parentFID, path)
            else if event_type is RENME then
                oldpath = cache.get(oldFID)
                newpath = cache.get(newFID)
                if oldFID not found in cache then
                    oldpath = fid2path(oldFID)
                    cache.set(oldFID, oldpath)
                if newFID not found in cache then
                    newpath = fid2path(newFID)
                    cache.set(newFID, newpath)
        catch fid2pathError:
            if event_type is UNLNK or RMDIR then
                path = ParentDirectoryRemoved
    resolvedEvent.add(event_type, time, date,
                      path/oldpath&newpath)
    return (resolvedEvent)
```

Processing (Ceph): Algorithm 5 shows the steps taken by the collector to process events from the Ceph Journal. Every event in the Journal is processed. As seen in Section 5.2.3, each event is associated with the event type, date and timestamp, dentry full and null bits, root_dentry full and null bits, and directory hash. The values for all of these parameters are extracted from each file system event. If the event is a *rename* event, both old and new paths are extracted from the null and full bits of dentry, respectively. If the event is a *delete* event, path is set as the null bits in dentry. If null bits in root_dentry exists, then, the event was a recursive deletion event. The directory hash value is also checked and *isDirectory* parameter is set as *True* if the hash value is non-zero.

Algorithm 5: Processing Ceph Journal events.

```
Input: Ceph File System Name $cephfs$, MDS Rank $mds$
Output: $EventList$
while $true$ do
    $events$ = read events from $mds$ Journal in $cephfs$
    for $event\ e\ in\ events$ do
        $resolvedEvent$ = processEvent($e$)
        $EventList$.add($resolvedEvent$)
    Clear Journal in $mds$
    return ($EventList$)
Function processEvent
    Input: Event $e$
    Output: $resolvedEvent$
    Extract $event\_type, ctime, dentry, root\_dentry, dirHash$ from $e$
    $path$ = $root\_dentry$.concat($dentry$)
    $isDirectory$ = False
    if $event\_type$ is rename then
        $oldpath$ = $root\_dentry$.concat($dentry\_nullbits$)
        $newpath$ = $root\_dentry$.concat($dentry\_fullbits$)
    else if $event\_type$ is unlink_local then
        $path$ = $root\_dentry$.concat($dentry\_nullbits$)
        if $root\_dentry$ has nullbits then
            $path$ = $ParentDirectoryRemoved$
    if $dirHash$ != $0$ then
        $isDirectory$ = True
    $resolvedEvent$.add($event\_type, ctime,$
                        $path/oldpath\&newpath, isDirectory$)
    return ($resolvedEvent$)
```

After events are processed by the collector, they are sent for aggregation.

Aggregation: Collectors use a publish-subscribe communication pattern (implemented with ZeroMQ [86]) to report events to an aggregator. When an event arrives to the aggregator it is placed in a processing queue. The aggregator service is multi-threaded, where one thread is responsible for publishing the aggregated file system events to the subscribed consumers, and the other thread stores the events into a local database to enable fault tolerance. This ensures minimal overhead on the system. An API is provided to the consumers to retrieve historic events from the database whenever a fault occurs. For our implementation, we use MySQL as the reliable event store.

Consumption: Consumers act as subscribers to the aggregator which publishes the events. The consumer service has the following three responsibilities.

Event filtering: Whenever a new event arrives to the consumer it filters the events and only passes on events related to those files and directories requested by the application. This filtering of events is not done at the aggregator in order to alleviate potential overheads if many consumers were to ask to monitor different files and directories.

Standard event representation: Before a new event is sent to the client application, event types are standardized according to the *inotify* format of file system event representation. This provides a generic monitoring solution for any large-scale storage system.

Fault tolerance: The consumer service is also responsible for retrieving events for a particular time stamp from the aggregator's reliable event store, in the situation that a consumer fails. Once retrieved, events are flagged as having been reported and can be removed from the data store when the next data purge cycle is initiated.

5.4 Evaluation

To evaluate the performance and overhead of `FSMonitor`, we have deployed `FSMonitor` on multiple Lustre parallel file system and Ceph object store platforms. Here we first describe these testbeds and the workloads that we use for evaluating `FSMonitor` performance, and then present our evaluation results.

5.4.1 Experimental Setup

We employ various testbeds to evaluate `FSMonitor` on both parallel file systems and object storage systems.

Parallel File System

We evaluate `FSMonitor` on three testbeds running Lustre.

AWS is a deployment of Lustre on five Amazon Web Service EC2 instances. We build a 20 GB Lustre file system using Lustre Intel Cloud Edition v 1.4 on five t2.micro instances and an EBS volume. Our Lustre configuration for AWS includes one MDS, one MGS, one OSS with one OST and two compute nodes.

Thor is a deployment at the Distributed Systems and Storage Laboratory (DSSL) at Virginia Tech. Each node in the setup has an 8-core processor, 16 GB memory and 512 GB storage. We deploy Lustre version 2.10.3 with one MGS, one MDS, seven OSSs each having five OSTs and two Lustre clients. Every OST is a 10 GB volume, resulting in a total of 350 GB Lustre store.

Iota is a production 897 TB Lustre deployment on a pre-exascale system at Argonne National Laboratory. This deployment has the same configuration and performance as the 150 PB store to be deployed on the Aurora supercomputer [2] at Argonne. The configuration includes Lustre's DNE with four MDSs. Iota has 44 compute nodes, each with 72 cores and 128 GB memory.

Object-based storage System

We evaluate `FSMonitor` on a *Ceph* testbed running Ceph version 10.2.11, with one admin node, one monitor, one client, one MDS, and two OSDs each with 4 TB object store. The MDS is placed in a metadata pool, and the two OSDs form one data pool. The admin and monitor nodes each have an 8-core Intel Xeon processor and 48 GB memory. The MDS and OSD nodes each have a 16-core Intel Xeon processor with 48 GB memory, and the Ceph client node has an 8-core Intel i7-6700 processor with 32 GB memory.

5.4.2 Experiment Workloads

We use a range of workloads to evaluate performance on the various testbeds. Specifically, we use one synthetic benchmark, *Evaluate_Performance_Script*; two real-world metadata benchmarks, *MDTest* and *FS_Mark*; and two real-world data benchmarks, *InterleavedOrRandom* and *FS_Mark*.

We run the synthetic benchmark on all three Lustre testbeds and the one Ceph testbed, and the metadata and data benchmarks on the Ceph testbed and *Thor* Lustre testbed. We run each experiment five times and report the mean values of the five runs.

Synthetic Benchmark

We use the *Evaluate_Performance_Script* synthetic benchmark to measure baseline performance. This benchmark repeatedly creates, modifies, and deletes a file *hello.txt*, in an infinite loop. This script thus tests `FSMonitor` efficiency over a period of time and can be used to evaluate `FSMonitor`'s resource utilization and event reporting rate. This script is also used as a baseline for event reporting in both Lustre file system and Ceph object store.

Metadata Benchmark

We use two metadata benchmarks to evaluate `FSMonitor` scalability and performance while reporting metadata events.

MDTest [13] measures the metadata performance of a file system. It works by creating, stating, and deleting a tree of directories and files in parallel across multiple clients in large scale storage systems. The MDTest benchmark generally exceeds application requirements for file systems and is therefore an appropriate benchmark to evaluate scalability and performance of `FSMonitor`. We run MDTest benchmark by varying the directory tree depth and the number of files. We ran MDTest five sets of values of files-per-directory: 300, 700, 3100, 11 100, and 42 100.

FS_Mark [8] tests synchronous write workloads, simulating workloads such as mail servers by performing multiple creations and writes of files across different directories. We ran FS_Mark for 1 million files.

Data Benchmark

In order to test the generality in event representation across parallel file systems and object stores, and to evaluate the impact on the I/O performance of applications under `FSMonitor`, we use two real-world data benchmarks.

The *InterleavedOrRandom* (IOR) [121] benchmark provides a flexible way to measure a distributed file system's I/O performance. It measures performance with different parameter configurations, including I/O interfaces ranging from traditional POSIX to advanced parallel I/O interfaces like MPI-I/O. It performs reads and writes to and from files on large scale storage systems, and calculates the throughput rates. For our evaluation of `FSMonitor`, IOR is executed with single shared file mode writing a 32 GB file with 8 processes.

The *HACC-I/O* is based on the *Hardware Accelerated Cosmology Code* (HACC) [120] application, which uses N-body techniques to simulate the formation of structure in collision-less fluids under the influence of gravity in an expanding universe. HACC-I/O captures the I/O patterns of the HACC simulation code. It uses the MPI-I/O interface and differentiates between FPP and SSF parallel I/O modes. We run HACC-IO for 10 million particles in file-per-process mode with 8 processes for our evaluation of `FSMonitor`.

5.4.3 Evaluating `FSMonitor` Using Synthetic Benchmark

We now explore `FSMonitor`'s performance when deployed on Lustre and Ceph using the synthetic benchmark *Evaluate_Performance_Script*.

Event capture rate

We first form a baseline throughput for Lustre and Ceph on each of the four testbeds: *AWS*, *Thor*, *Iota*, and *Ceph*. We use *Evaluate_Performance_Script* and record the number of events generated per second. As shown in Table 5.3, for Lustre, *AWS* performs the worst, as its Lustre testbed is formed from t2.micro instances. *Thor* performs better than *AWS* and, as expected, *Iota*'s performance is the best of the three. For Ceph, the event generation rates are similar to that of the *Thor* testbed for Lustre.

	Lustre			**Ceph**
	AWS	**Thor**	**Iota: 1 MDS**	
Storage Size	20 GB	350 GB	897 TB	8 TB
Create events/sec	352	746	1389	802
Modify events/sec	534	1347	2538	653
Delete events/sec	832	2104	3442	824
Total events/sec	1366	4509	9593	2086

Table 5.3: Baseline Event Generation Rates.

In our experimental setup, *AWS*, *Thor*, and *Ceph* have one MDS each, and *Iota* has four MDSs. However, for comparison purposes we present event generation rates in Table 5.3 using only a single MDS on all setups. As *AWS*, *Thor*, and *Ceph* have only one metadata catalog, `FSMonitor` extracts their events from only one MDS Changelog/Journal, processes them, and then communicates them to the aggregator for reporting to the client. On *Iota*, events are generated from all four MDSs and thus need to be collected from all the MDSs and then aggregated on the MGS before they are sent to the consumer.

Event Reporting Analysis

	Lustre			**Ceph**
	AWS	**Thor**	**Iota**	
Generated events/sec	1366	4509	9593	2086
*Reported events/sec **without** cache*	1053	3968	8162	2079
*Reported events/sec **with** cache*	1348	4487	9487	2079

Table 5.4: Baseline Event Reporting Rates.

When `FSMonitor` is deployed on Lustre, the resolution layer applies a caching mechanism to cache *fid2path* key-value pairs. To investigate the benefits of this approach we record the number of events captured per second with and without caching. Our results are shown in Table 5.4. We also include the baseline event generation rate from our previous experiment (Table 5.3).

For Lustre, when `FSMonitor` is used to report these events without caching the *fid2path* resolutions, the *AWS*-based `FSMonitor` can collect, process, and report only 1053 events per second. On *Iota*, it reports only 8162 events per second: 14.9% lower than the generation rate of ∼9500 events per second. When we analyze the `FSMonitor` event reporting pipeline, we notice that the performance is limited primarily in the processing of events after extracting them from the Lustre Changelog in the MDS. These results confirm that *fid2path* is costly and that execution for every event reduces overall throughput. In Ceph, we do not notice a difference in event reporting with or without cache, because `FSMonitor` on Ceph does not have to go through the *fid2path* resolution process.

For the implementation of `FSMonitor` on Lustre, we use an in-memory LRU cache to store the *fid2path* mappings. The size of the cache (number of *fid2path* mappings) is selected to be 5000 (we explore different cache sizes in Section 5.4.3). With the use of an in-memory cache, `FSMonitor` is able to to report 1348 events per second on *AWS*, 4487 events per second on *Thor*, and 9487 events per second on *Iota*.

The above results all use one MDS. On *Iota*, when we use all four available MDSs, the overall event generation rate is 38 372 events per second. `FSMonitor` is able to report 37 948 events per second to the consumer with caching enabled. Note that besides processing and filtering events, there is no additional overhead in the collection, aggregation, and reporting of events by `FSMonitor`. The difference between event reporting and event generation rates is due to the minimal processing required in `FSMonitor`. There is no loss of events; events are queued and simply processed at a slower rate than they are generated. As `FSMonitor` on Ceph requires much lesser processing of events than `FSMonitor` on Lustre, `FSMonitor`'s event loss rate is much lower for Ceph than Lustre.

Resource Utilization

We evaluate the resource utilization of every component of `FSMonitor`.

Evaluate_Performance_Script is used to perform this analysis. Tables 5.5 and 5.6 show the peak CPU and memory utilization respectively on each of our four testbeds. From the results, it is evident that the CPU and memory costs of operating `FSMonitor` are low.

We show collector's resource utilization both with and without a cache. The reduction in CPU utilization for `FSMonitor` in Lustre when caching is enabled is due to the lower number of *fid2path* invocations. For `FSMonitor` in Ceph, we see no difference in resource utilization due to caching because the *fid2path* invocations are not required in Ceph.

Next, we modified *Evaluate_Performance_Script* to create and delete, but not modify, files continuously. We noticed that the Collector service on *Iota* had a CPU usage of 3.3%: a 14.2% increase in CPU usage from 2.89%, when *Evaluate_Performance_Script* was tested. This is because delete events caused the *fid2path* mapping in the cache to fail for most events, resulting in *fid2path* calls on the parent directory. Memory usage did not change significantly.

We also changed *Evaluate_Performance_Script* to only create and modify, not delete, files. CPU usage in this case for *Iota* was 2.3%: a decrease of 20.4% from 2.89% when testing *Eval-*

uate_Performance_Script. This is because more frequent mappings in the cache were found. Even in this scenario, memory usage did not differ significantly. The resource utilization of the Aggregator and Consumer stays the same even when caching is enabled.

	CPU%			
	AWS	**Thor**	**Iota**	**Ceph**
Collector - No cache	9.3	7.8	6.7	1.4
Collector with cache	6.6	1.5	2.9	1.4
Aggregator	2.7	0.6	0.1	0.4
Consumer	1.5	0.2	0.1	0.1

Table 5.5: `FSMonitor` CPU Utilization.

	Memory (MB)			
	AWS	**Thor**	**Iota**	**Ceph**
Collector - No cache	8.2	33.7	81.6	10.5
Collector with cache	9.9	25.7	55.4	10.5
Aggregator	5.7	7.2	17.6	5.2
Consumer	0.1	0.2	2.8	0.2

Table 5.6: `FSMonitor` Memory Utilization.

Therefore, deploying `FSMonitor` on Lustre parallel file system (MDS, MGS and clients), and Ceph object store (MDS, Monitor and clients), would result in a negligible overload on the overall performance.

Caching

In order to calculate the optimum size for the in-memory LRU cache, we ran `FSMonitor` on *Iota* with *Evaluate_Performance_Script* and varied the cache size before every run. The results are shown in Table 5.7. We do not show the results from *Ceph*, because `FSMonitor` has negligible impact of caching in Ceph object store.

Cache Size (#fid2path)	CPU% on collector	Memory (MB) on collector	Events/sec reported by each collector
200	4.8	88.7	8644
500	3.5	84.3	8997
1000	3.0	75.6	9401
2000	3.0	61.3	9453
5000	2.9	55.4	9487
7500	2.9	60.7	9481

Table 5.7: `FSMonitor` performance vs. cache size.

The total number of events generated per second on *Iota* for one MDS is 9593, see in Table 5.3. As seen in Table 5.7, we observe improved performance using the in-memory LRU cache with size greater than or equal to 1000. For sizes 200 and 500, memory utilization on collector is even worse than than in `FSMonitor` without cache on *Iota*. The lowest CPU and memory utilization for Collector on *Iota* is for cache size 5000. Also, the number of events reported per second on one MDS is best for cache size of 5000. Increasing the cache size further to 7500 results in worse performance. Therefore, for our evaluation, we choose a cache size of 5000.

Comparison with Robinhood

There is no scalable file system event monitoring solution for object storage systems. However, for the Lustre parallel file system, there is a widely used monitoring tool, Robinhood [133], explained in Section 5.2.4. We compare `FSMonitor` with Robinhood on *Iota* with four MDSs.

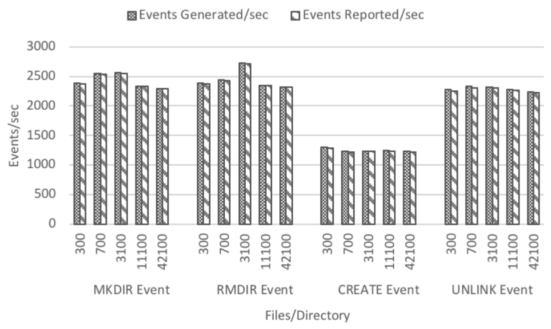
We implement Robinhood [107] by having a subscriber in the client that polls the four publishers on the MDS one at a time in a round-robin fashion. There is no role for MGS in this implementation. In comparison, `FSMonitor` has an aggregator service on MGS that polls all MDSs concurrently and pushes all events in a single queue to the clients. We evaluate performance by comparing the number of events per second received and processed at the client side by Robinhood vs. the number of events collected by the client via `FSMonitor` where the processing takes place at the MDSs and aggregation at the MGS. Our results show that Robinhood on *Iota* processes an average 7486 events per second from each MDS vs. 9847 events per second by `FSMonitor`. Combining all four MDSs, Robinhood processes 32 459 events per second in comparison to 37 948 events per second with `FSMonitor`. Therefore, with the advent of exascale supercomputers, where multiple MDSs on large-scale storage systems will be common, parallel monitoring is necessary.

5.4.4 Evaluating `FSMonitor` Using Metadata Benchmarks

We evaluate `FSMonitor` by running real-world metadata benchmarks *MDTest* and *FS_Mark*. For Lustre, we run the benchmarks on *Thor*.

Figure 5.6 shows the *MDTest* event generation rate and `FSMonitor` event reporting rate on Lustre and Ceph, respectively. We test scalability of `FSMonitor` by running *MDTest* from 300 files per directory to 42 100 files per directory. *MDTest* tests four event types: create directory (*MKDIR*), remove directory (*RMDIR*), create file (*CREATE*), and delete file (*UNLINK*). We see negligible difference in the event reporting rates, demonstrating `FSMonitor` scalability.

Figure 5.7 shows results for the *FS_Mark* metadata benchmark on Lustre and Ceph, respectively. *FS_Mark* evaluates metadata operations by creating (*CREATE*), writing (*MODIFY*), closing (*CLOSE*), and deleting (*UNLINK*) ten million files. As seen from the graphs, there is a negligible difference in the event reporting rate by `FSMonitor`, which is caused due to



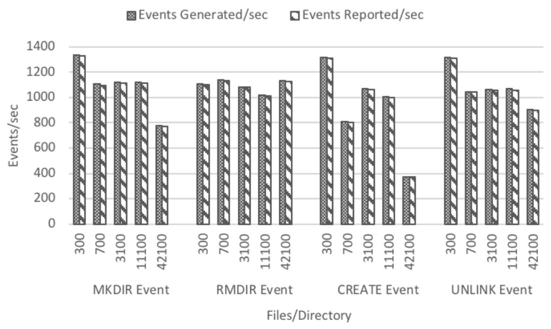
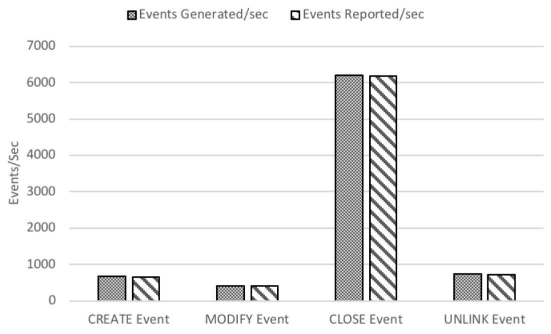


Figure 5.6: Performance of `FSMonitor` when running *MDTest* on Lustre (left) and Ceph (right).



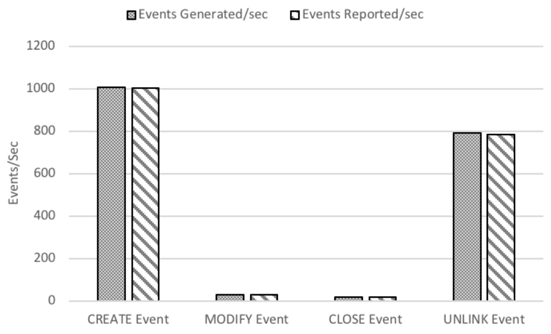


Figure 5.7: Performance of `FSMonitor` when running *FS_Mark* on Lustre (left) and Ceph (right).

the processing of events from the metadata catalog on the MDSs.

Thus, `FSMonitor` is scalable and is able to perform well on metadata stress testing.

5.4.5 Evaluating `FSMonitor` Using Data Benchmarks

We evaluate the impact of `FSMonitor` on large-scale HPC applications by running *HACC-I/O* and *IOR* on *Thor* and *Ceph*. We run both workloads simultaneously on the clients to see the impact of `FSMonitor` on concurrent running applications. We first evaluate the event representation on *Thor* and *Ceph*, which can be seen in Table 5.8. `FSMonitor` is able to provide the same event representation as *inotify* for both parallel file systems and object stores. As *IOR* was executed in single-shared-file mode, only single instances of *Create* and *Delete* file events were generated from IOR. *HACC-I/O* on the other hand was run in file-per-process mode for 8 processes. Therefore, 8 files were *created* and *deleted*. We did not notice any delay in the event reporting procedure by `FSMonitor` when the two workloads were executing simultaneously.

`FSMonitor` **events**
/mnt/lustre **CREATE** /hacc-io/FPP1-Part00000000-of-00000007.data
/mnt/lustre **CREATE** /hacc-io/FPP1-Part00000007-of-00000007.data
/mnt/lustre **CLOSE** /hacc-io/FPP1-Part00000000-of-00000007.data
/mnt/lustre **CLOSE** /hacc-io/FPP1-Par00000007-of-00000007.data
/mnt/lustre **CREATE** /ior/src/testFileSSF
/mnt/lustre **CLOSE** /ior/src/testFileSSF
/mnt/lustre **DELETE** /hacc-io/FPP1-Part00000000-of-00000007.data
/mnt/lustre **DELETE** /hacc-io/FPP1-Part00000007-of-00000007.data
/mnt/lustre **CLOSE** /hacc-io/FPP1-Part00000000-of-00000007.data
/mnt/lustre **CLOSE** /hacc-io/FPP1-Part00000007-of-00000007.data
/mnt/lustre **DELETE** /ior/src/testFileSSF
/mnt/lustre **CLOSE** /ior/src/testFileSSF

Table 5.8: `FSMonitor` events for IOR and HACC-IO on *Thor* and *Ceph*.

Next, we measure the impact of running *HACC-IO* with and without `FSMonitor`. The result is shown in Table 5.9. The read and write performance of *HACC-IO* are not impacted when `FSMonitor` is turned on in either the Lustre parallel file system or Ceph object store. We see similar results for *IOR*, as shown in Table 5.10.

HACC-IO	**without** `FSMonitor`		**with** `FSMonitor`	
	Lustre	**Ceph**	**Lustre**	**Ceph**
Read Bandwidth (MB/s)	1069.9	2157.8	1002.5	2085.5
Write Bandwidth (MB/s)	128.0	1005.2	138.9	1033.2

Table 5.9: Impact of running `FSMonitor` with HACC-IO.

IOR	**without** `FSMonitor`		**with** `FSMonitor`	
	Lustre	**Ceph**	**Lustre**	**Ceph**
Read Bandwidth (MB/s)	853.4	2686.4	871.1	2690.0
Write Bandwidth (MB/s)	239.4	575.3	272.4	616.2

Table 5.10: Impact of running `FSMonitor` with IOR.

5.5 An Illustrative Application Use Case

`FSMonitor` enables the development of various event-driven applications. As an illustrative example, we describe a file system indexer that uses `FSMonitor` to reduce re-indexing time.

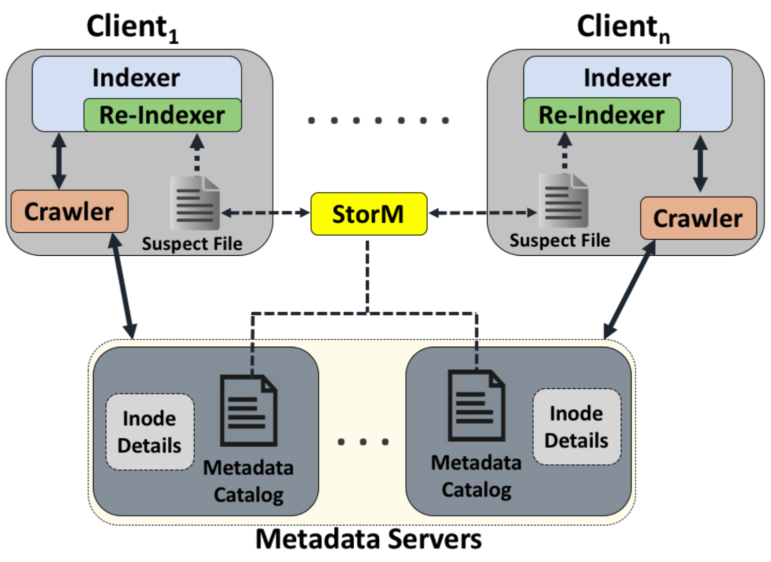


Figure 5.8: Overall design of file system indexer using `FSMonitor`.

As storage systems evolve to manage hundreds of petabytes and even exabytes of data, the cost of indexing the data is becoming increasingly prohibitive. Improved techniques are required to index file system data as hundreds of users concurrently create, modify, move, and delete data. Event-based indexing is necessary to enable flexible management of these extreme-scale stores.

Figure 5.8 shows the design of a file system indexer that uses `FSMonitor` to improve performance. The indexer and crawler are responsible for crawling the entire file system, collecting inode details from the metadata servers and indexing the file system metadata. This process takes hours for a petabyte scale file system. Therefore, there is a need for an effective re-indexer to update the indexed data. `FSMonitor` interacts with the metadata catalog to create a *suspect file* listing all directories on which file systems events occurred from a particular client. The re-indexer collects the list of these suspect directories periodically from the suspect file and indexes only these directories. We thus save the cost of crawling the entire file system to track changes.

#Files	**#Directories**	**Avg #Files per Dir**	**Total Size (GB)**
400 254 (400k)	43 189	9.3	4.4
1 200 762 (1.2M)	129 565	9.3	13.3
5 736 974 (5.7M)	619 029	9.3	63.4
8 530 405 (8.5M)	815 753	10.5	95.1
22 013 970 (22M)	2 375 341	9.3	243.2

Table 5.11: File system workload to evaluate re-indexer.

We implement the re-indexer using `FSMonitor` atop Brindexer [159], the indexer developed by Cray for the Clusterstor [6] storage platform for exascale systems. The evaluation is done on the *Thor* Lustre file system testbed described in Section 5.4.1. We use the file system load shown in Table 5.11, with the number of files ranging from 400 000 (4.4 GB) to 22 million (243.2 GB).

Table 5.12 shows the time taken by the indexer and re-indexer to index the file system. We run a script to modify a subset of files in random directories and then run the re-indexer to index the file system. As seen from the table, re-indexer, built atop `FSMonitor` is able to improve the re-indexing time by 93% for 22 million files, without having to crawl through the entire file system to track changes.

#Files	**Time taken by indexer (sec)**	**Re-indexing time without using `FSMonitor` (sec)**	**Re-indexing time using `FSMonitor` (sec)**
400k	155.2	18.0	8.5
1.2M	418.6	46.4	36.4
5.7M	1956.5	255.5	55.5
8.5M	3405.6	322.0	62.0
22M	9020.6	1075.2	75.5

Table 5.12: Time taken by indexer and re-indexer.

5.6 Chapter Summary

We have presented `FSMonitor`, a generic and scalable file system monitor for capturing and reporting events on heterogeneous large-scale storage systems. `FSMonitor` uses a three-layer approach to file system event monitoring. The lowest layer, *DSI*, interacts with a file system to detect events and sends them to the middle layer, *Resolution*. Here events are resolved to their absolute path names and aggregated to be sent to the upper layer, *Interface*. The Interface layer stores aggregated events which can be accessed by clients via the `FSMonitor` API.

`FSMonitor` implements a standard event definition process for any file system, and works seamlessly for both parallel file systems and object stores. We evaluated `FSMonitor` on three Lustre file system testbeds and a 8 TB Ceph store. We found that on a 897 TB Lustre system, `FSMonitor` reported almost 38 000 events per second with low resource utilization. On the 8 TB Ceph system, `FSMonitor` reported more than 2000 events per second. Compared to iterative monitoring methods used by the popular Robinhood system, `FSMonitor` achieves a 14.5% improved event reporting rate for multiple Lustre MDSs. `FSMonitor` also performs well for metadata benchmarks. We also did not notice any performance degradation in data benchmarks when using `FSMonitor`. Finally, we demonstrated order-of-magnitude improvements in large-scale file system re-indexing times when using `FSMonitor`.

Chapter 6

Efficient Metadata Indexing for HPC Storage Systems

6.1 Introduction

From life sciences and financial services to manufacturing and telecommunications, organizations are finding that they need not just more storage, but high-performance storage to meet the demands of their data-intensive workloads. This has resulted in a massive amount of data generation (order of petabytes), creation of billions of files, and thousands of users acting on HPC storage systems. According to a recent report from National Energy Research Scientific Computing Center (NERSC) [14], over the past 10 years, the total volume of data stored at NERSC has grown at an annual rate of 30 percent. This ever-increasing rate of data generation combined with the scale of HPC storage systems make efficiently organizing, finding, and managing files extremely difficult.

HPC users and system administrators need to query the properties of stored files to efficiently manage the storage system. This data management issue can be addressed by an efficient search of the file metadata in a storage system [111]. Metadata search is particularly helpful because it not only helps users locate files but also provides database-like analytic queries over important attributes. Metadata search involves indexing file metadata such as inode fields (for example, size, owner, and timestamps) and extended attributes (for example, document title, retention policy, and provenance), represented as <*attribute, value*> pairs [109]. Therefore, metadata search can help answer questions like *"Which application's files consume the most space in the file system?"* or *"Which files can be moved to second tier storage?"*.

Metadata indexing on large scale HPC storage systems presents a number of challenges. *First, scaling metadata indexing technology from local file systems to HPC storage systems is very difficult.* In local file systems, the metadata index has to index only a million files, and thus can be kept in-memory. However, in HPC systems, the index is too large to reside in-memory. *Second, the metadata indexing tool should be able to gather the metadata quickly.* The typical speed for file system crawlers is in the range of 600 to 1,500 files/sec [42]. This translates to 18 to 36 hours of crawling for a 100 million file data set. A large scale HPC storage system can often contain a billion files, which implies crawl time in the order of weeks [42]. *Third, the resource requirements should be low.* Existing HPC storage system metadata indexing tools such as LazyBase [62] and Grand Unified File-Index (GUFI) [9] require dedicated CPU, memory, and disk hardware, making them expensive and difficult to integrate into the storage system. *Fourth, metadata changes must be quickly re-indexed to prevent a search from returning inaccurate results.* It is difficult to keep the metadata index consistent because collecting metadata changes is often slow [181] and therefore, search applications are often inefficient to update.

Current state-of-the-art metadata indexing techniques on HPC storage systems include Spyglass [111], SmartStore [90], Security Aware Partitioning [146], and GIGA+ [149]. All of these techniques use a spatial tree, such as k-d tree [210], or R-tree [82] to index metadata. However, both these trees have poor performance in handling high dimensional data sets [40], they handle missing values inefficiently, and do not perform well for data which have multiple values for one field [184]. These drawbacks reduce their ability to index metadata efficiently. Other metadata indexing techniques, like, GUFI [9], Robinhood Policy Engine [107], and BorgFS [3], use a popular approach for metadata indexing where an external database is maintained for indexing outside the HPC storage system. This approach involves a major issue of maintaining consistency because the metadata is managed outside the file system which is being indexed.

To address these issues in HPC storage system metadata indexing, we present an efficient and scalable metadata indexing and search system, Brindexer. Brindexer enables a fast and scalable indexing technique by using a *leveled partitioning approach* to the file system. Leveled partitioning is different and more effective than the *hierarchical partitioning approach* used in state-of-the-art indexing techniques discussed above. Brindexer uses an in-tree indexing design and thus mitigates the issue of maintaining metadata consistency outside the file system. Brindexer also uses RDBMS to store the index which makes querying easier and more effective. To overcome the drawback of slow re-indexing process, Brindexer uses a changelog-based approach to keep track of metadata changes in the file system.

We present Brindexer and the scalable metadata changelog monitor that helps track the metadata changes in HPC storage system. The HPC storage system that we choose for our implementation is Lustre. According to the latest Top 500 list [16], Lustre powers $\sim 60\%$ of the top 100 supercomputers in the world. While the implementation and evaluation for Brindexer is shown in the chapter as applied to Lustre storage system, its design makes it applicable to other HPC storage systems, such as IBM's Spectrum Scale, GlusterFS and BeeGFS. We compare indexing and querying performance of Brindexer with the state-of-the-art GUFI indexing tool and show that the indexing performance of Brindexer is

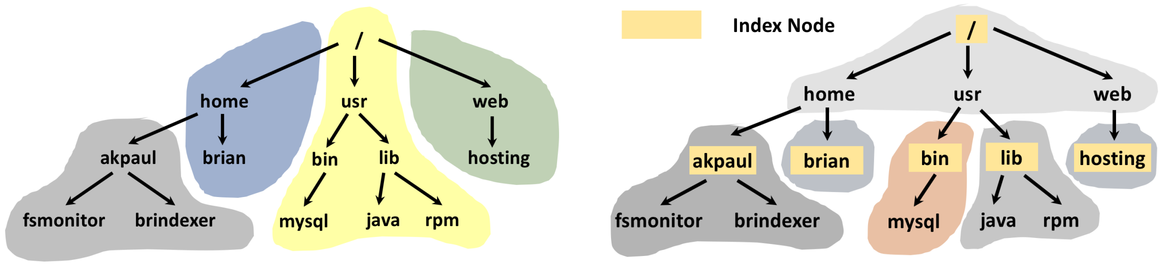


Figure 6.1: Comparison between hierarchical partitioning (left) and leveled partitioning (right) approaches using the same file structure.

better by 69%, querying performance is better by 91%. Resource utilization by Brindexer is lower than than of GUFI by 46% during indexing and 58% during querying 22 million files on a 4.8 TB Lustre store.

6.2 Background & Motivation

In this section, we describe the different partitioning approaches for indexing a file system, the metadata attributes motivated by some examples of file system search queries, the architecture of HPC storage system with an emphasis on Lustre file system, and finally we explain the different approaches to collecting metadata changes along with a motivation for Brindexer to use changelog-based approach.

6.2.1 Partitioning Techniques

To exploit metadata locality and improve scalability, HPC storage system's indexing tools partition the file system namespace into a collection of separate, smaller indexes. There are two main approaches to partitioning.

Hierarchical Partitioning

This is one of the most common approaches used in state-of-the-art metadata indexing tools. Hierarchical partitioning is based on the storage system's namespace and encapsulates separate parts of the namespace into separate partitions, thus allowing more flexible, finer grained control of the index. An example of hierarchical partitioning is shown in Figure 6.1. As seen in the figure, the namespace is broken into partitions that represent disjoint sub-trees. However, hierarchical partitioning faces an important challenge when the disjoint sub-trees are skewed, that is, some trees have more files than others.

Leveled Partitioning

This approach creates index nodes at a particular level in the storage system tree. An example of leveled partitioning is shown in Figure 6.1. In the figure, leveled partitioning is done at level 2. Therefore, the file system namespace is divided into disjoint sub-trees from level 2, with index nodes at the root of each sub-tree. This mitigates the issue of hierarchical partitioning where some trees may be skewed which affects indexing performance. In the leveled approach, all directories up to the next index level are indexed at the root of the current level. Another major issue of hierarchical partitioning is that a file system crawler should be used before indexing to partition the file system namespace into uniformly-sized disjoint sub-trees. This requires extra resource consumption which can be overcome by leveled partitioning where no such crawler is needed before indexing. BRINDEXER uses the leveled partitioning approach to partition the file system namespace into smaller indexes.

6.2.2 Metadata Attributes

File metadata can be of two types.

- *Inode Fields:* They are generated by the storage system itself for every file, and are shown in Table 6.1.
- *Extended Attributes:* These are typically generated by the users and applications. These may include *mime type* attribute, which defines the file extensions, and *permission* attribute specifying the read, write and execute permissions set by the application.

All attributes are typically represented in <*attribute, value*> pairs that describe the properties of a file. For each POSIX file there will be at least 10 attributes, and for a large scale HPC storage system with a billion files, there will be a minimum of 10^{10} attribute pairs. The ability to search this massive dataset of metadata attributes pairs effectively gives rise to metadata indexing.

Attribute	**Description**		**Attribute**	**Description**
ino	inode number		*size*	file size
mode	file or directory		*blocks*	blocks allocated
nlink	number of hard links		*atime*	access time
uid	owner of file		*mtime*	modification time
gid	group owner of file		*ctime*	status change time

Table 6.1: Metadata Attributes.

Some common metadata attributes used are shown in Table 6.1. The attribute *mode* is used with the proper mask macros (*S_IFREG* and *S_IFDIR*) to determine if a file is a regular

Storage System Administrator Question	**Metadata Search Query**
Which files should be migrated to secondary storage?	*size* >100 GB, *atime* >1 year
Which files have expired their legal compliances?	*mode* = file, *mtime* >10 years
How much storage do each user consume?	Sum *size* where *mode* = file, group by *uid*

Table 6.2: Some sample file management questions and the metadata search queries used.

file or a directory. *atime* attribute is affected when a file is handled by *execve*, *mknod*, *pipe*, *utime*, and *read* (of more than zero bytes) system calls. *mtime* is affected by the *truncate* and *write* calls. *ctime* is changed by writing or by setting inode information.

Some sample file management questions and the queries used to search the metadata attributes are shown in Table 6.2. These show the importance of fast and scalable metadata indexing and querying that can help HPC storage system administrators.

6.2.3 HPC Storage System

HPC storage systems are designed to distribute file data across multiple servers so that multiple clients can access file system data in parallel. Typically, they consist of *clients* that read or write data to the file system, *data servers* where data is stored, *metadata servers* that manage the metadata and placement of data on the data servers, and networks to connect these components. Data may be distributed (divided into stripes) across multiple data servers to enable parallel reads and writes. This level of parallelism is transparent to the clients, for whom it seems as though they are accessing a local file system. Therefore, important functions of a distributed file system include avoiding potential conflicts among multiple clients and ensuring data integrity and system redundancy. The most common HPC file systems include Lustre, GlusterFS, BeeGFS, and IBM Spectrum Scale. In this chapter, we have built Brindexer on the Lustre file system.

Lustre File System

The architecture of the Lustre file system is discussed in Chapter 3 [153, 196]. Lustre has a client-server network architecture and is designed for high performance and scalability. The *Management Server (MGS)* is responsible for storing the configuration information for the entire Lustre file system. This persistent information is stored on the *Management Target (MGT)*. The *Metadata Server (MDS)* manages all the namespace operations for the file system. The namespace metadata, such as directories, file names, file layout, and access permissions are stored in a *Metadata Target (MDT)*. Every Lustre file system must have a minimum of one MDT. *Object Storage Servers (OSSes)* provide the storage for the file contents in a Lustre file system. Each file is stored on one or more *Object Storage Target (OST)*s mounted on the OSS. Applications access the file system data via *Lustre clients* which interact with OSSes directly for parallel file accesses. The internal high-speed data networking protocol for the Lustre file system is abstracted and is managed by the *Lustre Network (LNet)* layer.

6.2.4 Collecting Metadata Changes

After metadata indexing is done, regular re-indexing needs to be performed so that metadata search queries do not return out-of-date results. Re-indexing of the metadata can be performed by running the indexing tool at regular intervals to index the entire file system afresh. This is an approach that most state-of-the-art indexing techniques (GUFI [9], and BorgFS [3]) use which maintain the index in an external database outside the file system. However this is a very expensive approach for large filesystems. Another approach is to keep track of metadata changes and re-index based on the changes. There are two ways to collect metadata changes: *Snapshot-based approach* and *Changelog-based approach*.

- *Snapshot-Based Approach:* In this approach periodic snapshots are taken of the file system metadata. Snapshots are created by making a copy-on-write (CoW) clone of the inode file. Given two snapshots at time instant T_n and T_{n+1}, this approach will calculate the difference between these two snapshots and identify the files that have changed during the time interval between the two snapshots. The metadata index crawler can only crawl over the changed files to re-index them. This is much faster than periodic walks of the entire file system. However, this approach depends on a filesystem design incorporating CoW metadata updates.
- *Changelog-Based Approach:* This approach logs the metadata changes as the changes occur on the file system. This is done by recording the modifying events that occur on the file system. Every HPC storage system maintains an event changelog (used for auditing purposes), example *mmaudit* in IBM Spectrum Scale, and *Lustre Changelog* in Lustre file system. Thus, building a scalable monitor that monitors the changelog could be a very efficient solution for collecting metadata changes. Only the files on which any modification event occurs need re-indexing. In Brindexer, we use the changelog-based approach for collecting metadata changes.

Next, we explain the Lustre changelog which is used to keep track of file system events on Lustre file system.

Lustre Changelog

Table 6.3 shows sample records in Lustre's Changelog. We ran a simple script to see the events recorded in the Changelog. The script first creates a file, *hello.txt*, then the file is modified. The file is then renamed to *hi.txt*. A directory named *okdir* is then created. Finally, we delete the file.

ID	Type	Time	Date	Flags	Target FID	Parent FID	Target
11	01CREAT	21:18:47.30	2020.06.20	0x0	t=[0x5716:0x626c:0x0]	p=[0x5716:0xe7:0x0]	hello.txt
12	17MTIME	21:18:47.32	2020.06.20	0x7	t=[0x5716:0x626c:0x0]		hello.txt
13	08RENME	21:18:47.41	2020.06.20	0x1	t=[0x5716:0x17a:0x0]	p=[0x5716:0xe7:0x0] s=[0x5716:0x626b:0x0] sp=[0x5716:0x626c:0x0]	hello.txt hi.txt
14	02MKDIR	21:18:47.42	2020.06.20	0x0	t=[0x5716:0x626d:0x0]	p=[0x5716:0xe7:0x0]	okdir
15	06UNLNK	21:18:47.43	2020.06.20	0x0	t=[0x5716:0x626b:0x0]	p=[0x5716:0xe7:0x0]	hi.txt

Table 6.3: A sample Lustre ChangeLog record showing *Create File*, *Modify*, *Rename*, *Create Directory*, and *Delete File* events.

Each tuple in Table 6.3 represents a file system event. Every row in the Changelog has an *EventID* – the record number of the Changelog; *Type* – the type of file system event that occurred; *Timestamp, Datestamp* – the date time of the event occurrence; *Flags* – masking for the event; *Target FID* – file identifier of the target file/directory on which the event occurred; *Parent FID* – file identifier of the parent directory of the target file/directory; and the *Target Name* – the file/directory name which triggered the event. It is evident that the Parent and Target FIDs need to be resolved to their original names before they can be processed by Brindexer. The following events are recorded in the Changelog:

- `CREAT`: Creation of a regular file.
- `MKDIR`: Creation of a directory.
- `HLINK`: Hard link.
- `SLINK`: Soft link.
- `MKNOD`: Creation of a device file.
- `MTIME`: Modification of a regular file.
- `UNLNK`: Deletion of a regular file.
- `RMDIR`: Deletion of a directory.
- `RENME`: Rename a file or directory.
- `IOCTL`: Input-output control on a file or directory.
- `TRUNC`: Truncate a regular file.
- `SATTR`: Attribute change.
- `XATTR`: Extended attribute change.

Note in Table 6.3 that Target FIDs are enclosed within $t = []$, and parent FIDs within $p = []$. *MTIME* event does not have a parent FID. *RENME* event has additional FIDs, $s = []$ denoting a new file identifier to which the file has been renamed, and $sp = []$ gives the file identifier for the original file. These features are important when resolving FIDs.

Motivation for using Lustre Changelog

We analyze a 24-hour Lustre Changelog obtained from a production system's petascale Lustre file system in Los Alamos National Laboratory (LANL).

Some observations from the analysis are:

- There are more than 34 million file system events which occur per day in a large-scale production-level HPC storage system.
- The number of unique files that get affected in 24 hours is ∼ 10.5 million.
- The number of unique directories on which metadata events occur is ∼ 110,000.
- The number of events for each individual event is shown in Table 6.4.

Event Type	# Events		Event Type	# Events
CREAT	1,322,010		MKDIR	67,791
HLINK	8,841		SLINK	94,711
MTIME	10,098,485		UNLNK	750,480
RMDIR	59,841		RENME	97,227
SATTR	3,432,589		XATTR	164

Table 6.4: Number of file system events for each metadata event in a 24-hour Lustre Changelog.

The analysis shows that performing a snapshot-based approach for keeping track of metadata changes in the file system may be very expensive for large, active filesystems. Also, to determine the directories for the affected 10.5 million files is time-consuming. The Lustre changelog, however, already reports the parent directories, which are also the directories which need to be re-indexed. This will improve the performance of BRINDEXER immensely because it does not need to keep track of all the 10.5 million files for re-indexing, but only the 110,000 unique directories. The challenge is to design an efficient and scalable changelog processing engine to get the parent FIDs (directories) of more than 10 million *MTIME* and more than 3 million *SATTR* files which are not already recorded in the changelogs. This is discussed in Section 6.3.2.

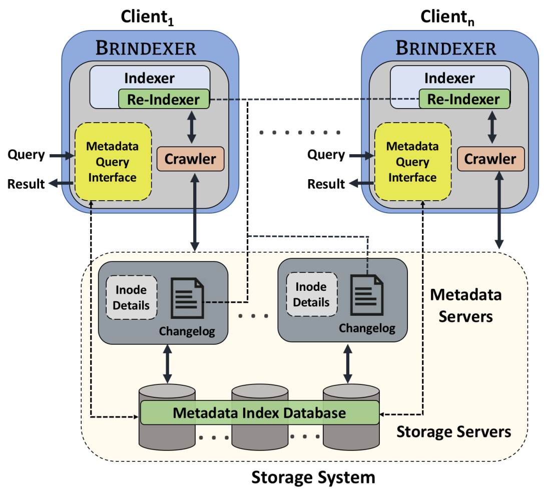


Figure 6.2: Overall architecture of Brindexer.

6.3 System Design

This section describes the design and implementation details of Brindexer. The overall architecture of Brindexer is shown in Figure 6.2. Brindexer runs on the file system clients. It consists of the indexer, crawler and the metadata query interface. The indexer and crawler are responsible for crawling the entire file system, collecting the inode details from the metadata servers and indexing the file system metadata. As Brindexer runs on the same file system which it indexes, the integrated metadata index database is created on the storage servers of the file system. The re-indexer is part of the indexer in Brindexer and interacts with the file system changelog to keep track of the metadata changes. Users and applications interact with the the metadata query interface provided by Brindexer to query the metadata index database. Next, we describe each component of Brindexer in more details.

6.3.1 Indexer

The overview of the indexing process of Brindexer is shown in Algorithm 6. Brindexer uses a leveled partitioning technique to partition the file system namespace. This is described earlier in Section 6.2. Brindexer performs the leveled partitioning approach in parallel, where the indexing task can be distributed on multiple client indexers for fast and scalable indexing. Each client node can be assigned a set of sub-trees and independently manages the file system indices under those sub-trees. This is shown in Figure 6.3 and this paralleled approach improves the performance of Brindexer. *Crawler* is responsible for doing the directory walk of the file system namespace.

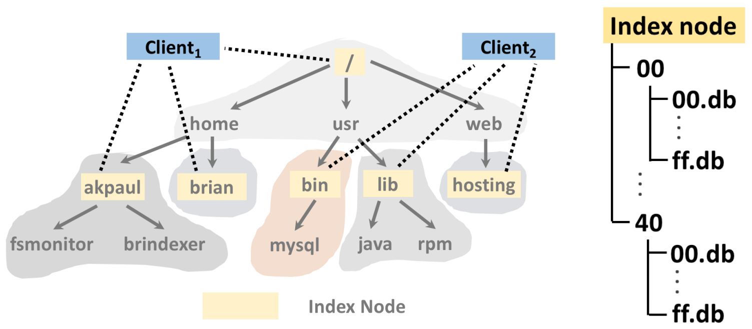


Figure 6.3: Parallelism in leveled partitioning (left) and 2-level database sharding (right) of BRINDEXER.

Algorithm 6: Indexing function in BRINDEXER.

```
Function Indexing
    Input: Indexing Level: indexLevel
    Output: Metadata Index Database: indexdb
    for Directory dir in directoryWalk do
        if dir in Level indexLevel then
            Setup database in Index Directory
            processIndexDir(dir)
    processRootDir(level)
    return (indexdb)
Function processIndexDir
    Input: Index Directory: dir
    for Directory subdir in recursive read of ll_readdir(dir) do
        hash = Calculate hash of subdir
        for File file in stat(subdir) do
            new_lstat(file)
            Place inode information of file in the database shard with the hash value as hash
Function processRootDir
    Input: Indexing Level: indexLevel
    Setup database in Root Directory
    for Directory dir in directoryWalk do
        if dir < Level indexLevel then
            for Directory subdir in recursive read of ll_readdir(dir) do
                hash = Calculate hash of subdir
                for File file in stat(subdir) do
                    new_lstat(file)
                    Place inode information of file in the database shard with the hash value as hash
```

The input to *Indexer* is the indexing level where all the directories at that level need to be indexed. The root directory is responsible to index all directories above the indexing level. For each indexed directory, a recursive *readdir()* is performed to find all sub-directories. For every sub-directory, a *stat()* call is made to get the files in that directory, and to get the inode information for every file, *new_lstat* call is performed on the file.

Each individual index directory is set to a 2-level database sharding approach to keep the database shards to a reasonable size. This is done to maximize the database performance by querying an optimum number of files per database. This is shown in Figure 6.3. The number of databases per index node is limited to 64 (0x40). This number is based on experiments to measure the time to index and query Brindexer for 1 billion files. 64 gives the optimum performance by having the optimal resource utilization. Within each database in the index node, there are database shards. Each database shard holds metadata information of one or more sub-directories of the index directory. To find the placement of the metadata information for a file, first MD5 hashing is done on the parent directory of the file to get the database shard. Next, MD5 hash is done on the index directory to find the database within which the database shard is placed. This 2-level sharding is done by Brindexer to maximize the query performance.

6.3.2 Re-Indexer

The architecture of *re-indexer* is shown in Figure 6.4.

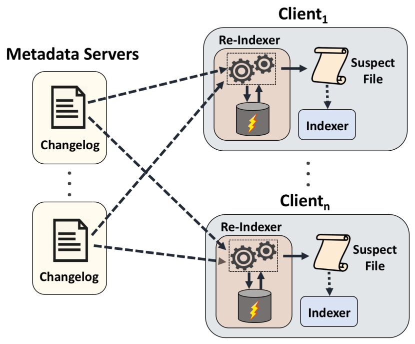


Figure 6.4: Design of Re-Indexer in Brindexer.

Brindexer's *re-indexer* is a multi-threaded set of processes running on filesystem clients. One thread is responsible for processing the file system changelogs gathered from the metadata servers, which are processed in parallel on the clients. A fast and efficient caching mechanism is used to store the mappings of FIDs to paths to improve performance of processing of the changelogs. Another thread maintains a suspect file which has a collection of all suspect directories (directories which have been modified and need to be re-indexed) for a particular time period. This suspect file is given as an input to the indexer which then does a *stat()* for only the suspect directories.

Processing Changelogs

The *re-indexer* collects events from changelog in batches. Every event that is collected needs to be processed to collect the directory name in order to be placed in the *suspect file*. In particular, FIDs are not necessarily interpretable by Brindexer, and thus must be processed and resolved to absolute path names. In Lustre file system, to process the FIDs, Lustre *fid2path* tool is provided which resolves FIDs to absolute path names. However, the *fid2path* tool is slow and can delay the reporting of events. For example, in Section 6.4.6 we show that this delay can cause a decrease of 31.7% in the event reporting rate compared to the events generated in the file system. To minimize this overhead, *re-indexer* implements a Least Recently Used (LRU) Cache to store mappings of FIDs to source paths.

Algorithm 7: Processing Changelog events in Lustre file system.

```
Input: Lustre path $lpath$, Cache $cache$, MDT ID $mdt$
Output: $SuspectFile$
while $true$ do
    $events$ = read events from $mdt$ Changelog
    for $event\ e\ in\ events$ do
        $resolvedPath$ = processEvent($e$)
        $SuspectFile$.add($resolvedPath$)
    Clear Changelog in $mdt$
    return ($SuspectFile$)
Function processEvent
    Input: Event $e$
    Output: $resolvedPath$
    Extract $event\_type, time, date$ from $e$
    try:
        $path$ = $cache$.get($parentFID$)
        if $parentFID$ not found in cache then
            $path$ = $fid2path$($parentFID$)
            $cache$.set($parentFID$, $path$)
    catch $fid2pathError$:
        $path$ = $cache$.get($targetFID$)
        if $targetFID$ not found in cache then
            $path$ = $fid2path$($targetFID$)
            Remove file name from path
            $cache$.set($targetFID$, $path$)
    return ($path$)
```

Algorithm 7 shows the processing steps for *re-indexer*. Changelog events are processed in batches. A LRU cache is used to resolve parent FIDs (directories) to absolute paths. Whenever an entry is not found in the cache, we invoke the *fid2path* tool to resolve the FID and then store the mapping *(FID – path)* into the LRU cache. `MTIME` and `SATTR` events do not have a parent FID and thus they are processed in the catch block, where the target FIDs are processed. The file name from the absolute path is removed to get the directory name and then the path is added to the cache, so that *fid2path* tool is not called on the file again. It should be noted that the cache only needs to track modified parent directories, so only 110,000 entries are present in a 24-hour suspect file rather than 10.5 million files. All of the resolved directory paths are added to the *suspect file* (not adding duplicates). After processing a batch of file system events from the Changelog, *re-indexer* will purge the Changelogs. A pointer is maintained to the most recently processed event tuple and all previous events are cleared from the Changelog. This helps reduce the overburdening of the Changelog with stale events.

Indexer periodically reads the suspect file and re-indexes the file system based on the suspect directories. Once *indexer* acts on a suspect file, a timestamp is given to the *re-indexer* and a new suspect file is written to add suspect directories from that time stamp.

6.3.3 Metadata Query Interface

Metadata query interface in Brindexer interacts with the *metadata index database* which is stored on the storage servers in the file system. The *metadata index database* uses RDBMS to store the index information of large-scale HPC storage systems. There are few reasons for selecting RDBMS for our implementation. First, we are not concerned with scalability of the database because our design of *indexer* and *re-indexer* takes care of it. We use parallel leveled partition approach for speed and 2-level database sharding in the index level directory for scalability and optimal query performance. RDBMS therefore serves its purpose of providing a nice API for the users to query the database. Second, RDBMS is very efficient in handling bulk writes and appends which is needed during the re-indexing process. Also, doing bulk reads on RDBMS is efficient. Third, the limitation of RDBMS is when it has to handle continuous stream of inputs. In Brindexer, the *metadata index database* only has periodic input stream and RDBMS works efficiently in this case. Fourth, RDBMS also lowers performance when it has to handle contended writes as it has to deal with multiple locking issues. The 2-level sharding and the namespace partition to handle disjoint sub-trees in Brindexer does not involve *metadata index database* to handle contended writes.

6.4 Evaluation

We evaluate the performance of Brindexer by analyzing each component in detail. In this section, we describe the the experimental setup for the evaluation, workloads that were used for analyzing the performance, and evaluate *indexer*, *re-indexer*, and *metadata query interface*.

6.4.1 Experimental Setup

To evaluate Brindexer, we use a Lustre file system cluster of 9 nodes with 4 MDSs, 3 OSSs and 2 clients. All nodes run CentOS 7 atop a machine with an AMD 64-core 2.7 GHz processor, 128 GB of RAM, and a 2.5 TB SSD. All nodes are interconnected with 10 Gbps bandwidth ethernet. Each MDS has a 128GB MDT associated with it. Furthermore, each OSS has 3 OSTs, with each OSS supporting 1.6 TB attached storage on OSTs. Therefore, our analysis is done on a 4.8 TB Lustre store.

6.4.2 Workloads

We use 2 kinds of workloads to test the performance of Brindexer as shown in Tables 6.5 and 6.6.

#Files	#Directories	Avg #Files per Dir	Total Size (MB)
400,000 (400k)	1,000	400	1.08
1,200,000 (1.2M)	1,000	1200	43.05
5,700,000 (5.7M)	5,700	1000	90.07
8,500,000 (8.5M)	8,500	1000	140.78
22,000,000 (22M)	222,000	1000	373.9

Table 6.5: Workload 1: Flat directory structure: Smaller number of directories with higher average number of files per directory.

#Files	#Directories	Avg #Files per Dir	Total Size (GB)
400,254 (400k)	43,189	9.26	4.4
1,200,762 (1.2M)	129,565	9.27	13.3
5,736,974 (5.7M)	619,029	9.27	63.4
8,530,405 (8.5M)	815,753	10.46	95.1
22,013,970 (22M)	2,375,341	9.27	243.2

Table 6.6: Workload 2: Hierarchical directory structure: Large number of directories with lower average number of files per directory.

Both workloads have 5 different numbers of files (400k, 1.2M, 5.7M, 8.5M, and 22M). However, the workloads differ in the number of directories. Workload 1 has a flat directory structure with just one level consisting of lower number of directories with higher number of files per directory. Workload 2 has a hierarchical structure (with maximum directory depth of 17) consisting of higher number of directories with lower number of file per directory. Therefore, workload 1 represents ideal case while workload 2 takes care of the real-world use case.

We compare Brindexer with state-of-the-art indexing tool GUFI [9] for indexing. For workloads 1 and 2, Brindexer sets an indexing level of 1 and 3 respectively. This is

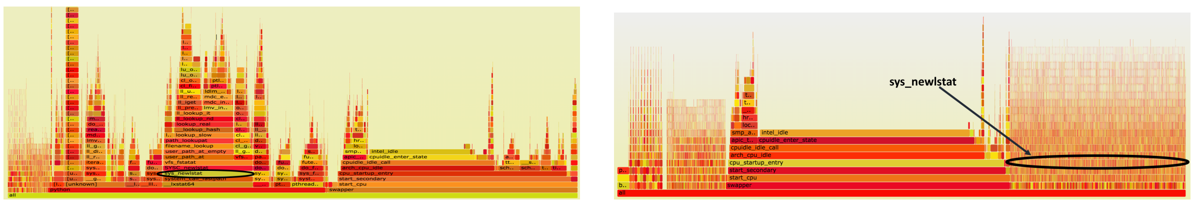


Figure 6.5: Comparison of system call stack for indexing 1.2M files in hierarchical directory structure by Brindexer (left) and GUFI (right).

because, workload 1 has only one level, and for workload 2, it turns out that number of directories at level 3 equals the average number of directories per level.

To evaluate querying performance of Brindexer, besides GUFI, we also compare Brindexer with Lustre's default *lfs find* tool [11] and *Robinhood policy engine* [107].

For evaluation of *re-indexer*, we evaluate the performance of Lustre changelog processing and compare it with *Robinhood* [107], as these are the only tools which use changelog-based approach to keep track of metadata changes.

All experiments are run five times and the evaluation shows the average of these runs. All caches are cleared between runs.

Next we give a brief description of GUFI and Robinhood.

GUFI stands for Grand Unified File Index. It uses breadth first traversal to traverse the entire file system tree in a parallel manner. GUFI uses one index database per directory to have the same security permission as that of the directory. This entire database tree is done outside the file system. Although GUFI is meant for indexing into an external database, we modify GUFI to run in-tree and perform metadata indexing inside Lustre file system itself. This provides a much fairer comparison with Brindexer. Brindexer is not concerned with directory permissions because it is meant for system administrators.

Robinhood collects information from the filesystem it monitors and inserts this information into an external database. It makes use of the Lustre changelog to monitor and keep track of file system events. For multiple MDSs, Robinhood uses a round-robin approach to keep track of changelogs.

6.4.3 Comparison of System Calls

Both Brindexer and Gufi were used to index 1.2 million files in hierarchical directory structure and the system calls were traced in a CPU flame graph [7]. This is shown in Figure 6.5. In a flame graph, each box represents a function in the stack. On the y-axis, the depth of the stack is shown and x-axis spans the sample population. The width of each box shows the amount of time a system call spends on CPU. The major observation from the flame graphs is that *sys_newlstat()* which is used for getting a file's inode information is represented as one block in Brindexer and multiple individual stack calls in GUFI. Also, cumulative width of the boxes for *sys_newlstat()* in GUFI exceeds the width in Brindexer, which means that GUFI spends more time in CPU for retrieving file information. This shows that Brindexer is more effective in using the system call to retrieve file information than GUFI.

6.4.4 Evaluation of Indexer

Time Taken to Index

The time taken to index both workloads by Brindexer and GUFI is shown in Figure 6.6. As seen in the figure, GUFI performs better than Brindexer for workload 1 where there is a flat directory structure. This is because the design of GUFI enables optimization for a directory level of just 1 where each directory has a large number of files. However, for the real world case in workload 2, Brindexer outperforms GUFI. As the number of files increase, the time to index in GUFI increases exponentially, with the maximum difference between Brindexer and GUFI of 69% in the time taken to index seen for 22M files.

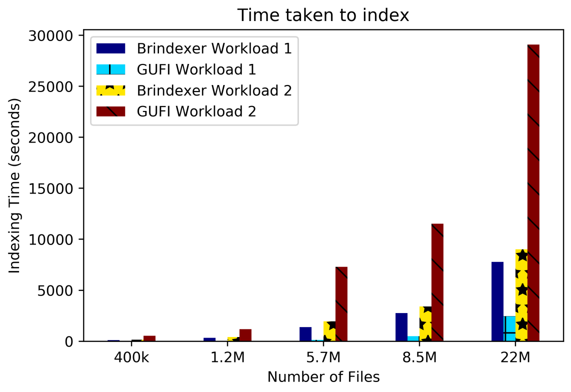


Figure 6.6: Time taken to index by Brindexer and GUFI.

Resource Utilization

Figures 6.7 and 6.8 show the resource utilization of Brindexer and GUFI when indexing the file system for both workloads. The legend used in Figure 6.7 is consistent for all the graphs. We only show the resource utilization of Lustre client and MDS. The behavior shown is similar on the OSSs. The CPU utilization of Brindexer during indexing is lesser than GUFI by 46.6% on clients and 86.04% on the MDSs. It can be further seen that GUFI is much more CPU intensive than Brindexer on MDS. This is because of the multiple individual

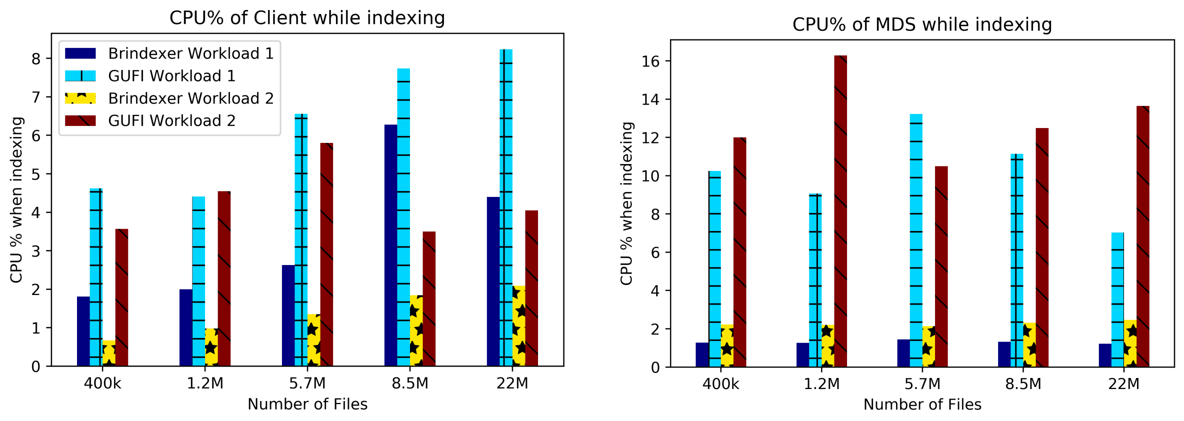


Figure 6.7: CPU utilization by Brindexer and GUFI during indexing on Client (left) and MDS (right).

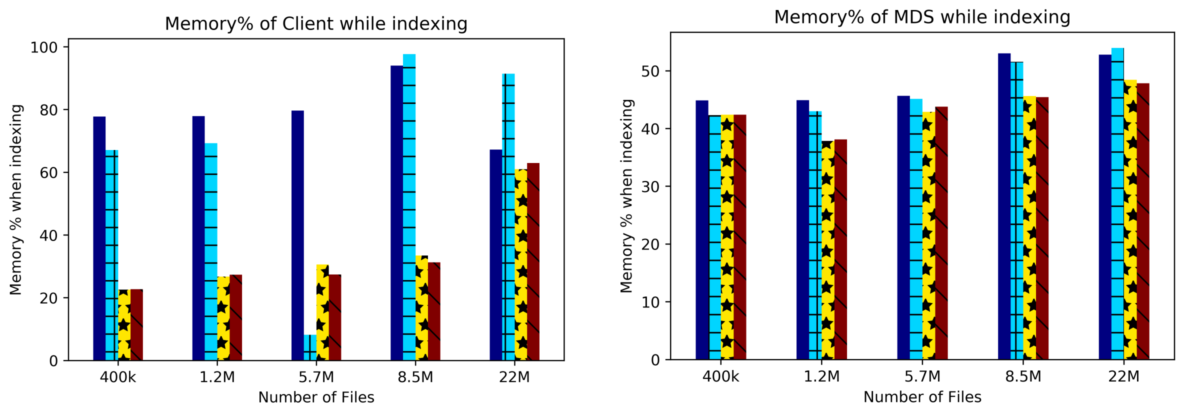


Figure 6.8: Memory utilization by Brindexer and GUFI during indexing on Client (left) and MDS (right).

stat calls that GUFI makes to the MDS as seen in Figure 6.5. Even for workload 1, where GUFI takes less time to index than Brindexer, the CPU utilization is much more than Brindexer. Similar behavior as CPU is shown on other resources like network and disk. However, in case of memory, both Brindexer and GUFI have a similar memory utilization on clients and MDS as seen in Figure 6.8.

6.4.5 Evaluation of Metadata Query Interface

To evaluate the *metadata query interface* of Brindexer, we run a query to find all files whose size is greater than 10 MB. We compare the query performance of Brindexer with GUFI, Lustre's *lfs find* tool, and *Robinhood policy engine*.

Time Taken to Query

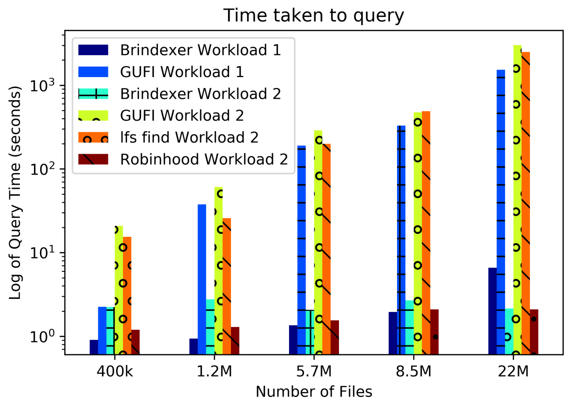


Figure 6.9: Time taken to query by BRINDEXER, GUFI, *lfs find*, and Robinhood.

Figure 6.9 shows the time taken to run the query and get the results back from the index database from BRINDEXER, GUFI, lfs find, and Robinhood. GUFI performs worse than BRINDEXER for both the workloads. This is because BRINDEXER makes use of parallel search on all the index nodes. The 2-level database sharding in BRINDEXER helps optimize queries further. We compare BRINDEXER with lfs find and Robinhood using queries on only workload 2. *lfs find* traverses the entire file system to get the results without using indexing and performs the worst. The difference in query performance between BRINDEXER and Lustre's default lfs find tool is proportional to the number of index nodes at the indexing level. The query performance of BRINDEXER and Robinhood is similar, though Robinhood uses an external database to index the file system. Therefore, BRINDEXER reaches an ideal query performance in the file system itself and improves upon state-of-the-art GUFI by 91%.

Resource Utilization

Figures 6.10 and 6.11 show the resource utilization of BRINDEXER and GUFI when querying the file system for both workloads. The legend used in Figure 6.10 is consistent for all the graphs. The resource utilization during querying is shown on OSSs instead of MDS because metadata index database resides in the OSSs of the file system. It is seen that GUFI's query task is much more CPU intensive than that of BRINDEXER. The memory utilization is similar for both. Therefore, BRINDEXER helps reduce CPU utilization during queries by 91.8% on clients and 57.8% on OSSs compared to GUFI.

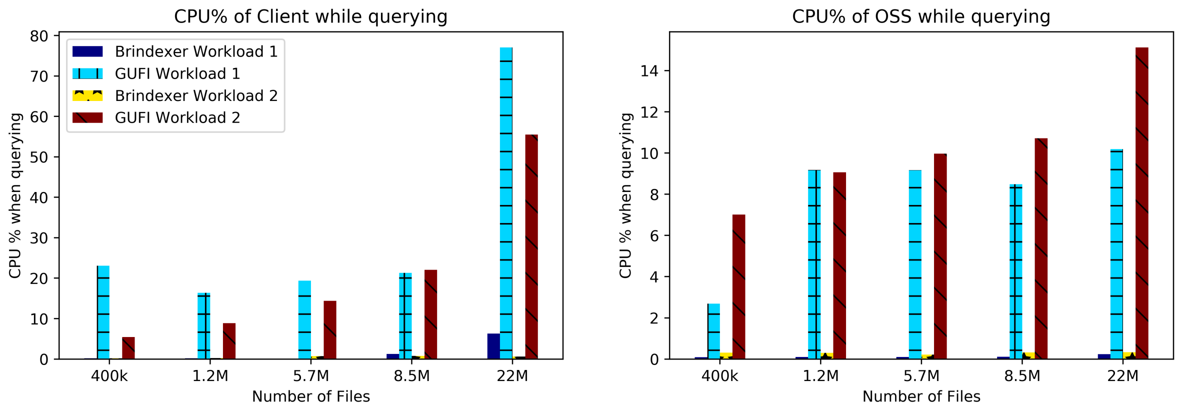


Figure 6.10: CPU utilization by Brindexer and GUFI during querying on Client (left) and MDS (right).

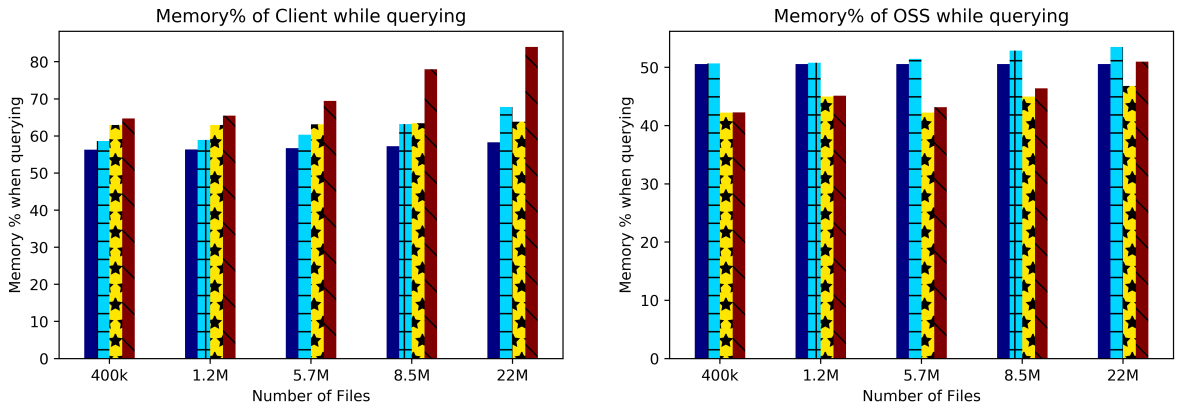


Figure 6.11: Memory utilization by Brindexer and GUFI during indexing on Client (left) and MDS (right).

6.4.6 Evaluation of Re-Indexer

The analysis of 24-hour Lustre changelog that was described in Section 6.2.4 shows that on a large scale production level Lustre store, more than 34 million events occur per day which corresponds to ∼400 events per second. We write a script that operates on the 22 million file dataset in workload 2 and generates 766 random events (create, modify and remove) per second per MDS. We then evaluate the performance of re-indexer in reporting these events.

Event Reporting Analysis

The event reporting rates (rate at which the suspect file is created) of Brindexer and Robinhood are shown in Table 6.7. Lustre's *fid2path* tool is resource intensive and slow. Therefore, there is a 28.7% improvement in event reporting rate when LRU cache is used in BRINDEXER's re-indexer to save parent FID and path mappings. BRINDEXER performs better than Robinhood when it come to event reporting comparison because of Robinhood's round-robin approach to processing changelogs from the MDSs. BRINDEXER uses a parallel and scalable approach which improves the event reporting rate.

	#Events per second
Events generated	766
Events reported by BRINDEXER *without cache*	523
Events reported by BRINDEXER *with cache*	734
Events reported by Robinhood	710

Table 6.7: Event Reporting Rates by BRINDEXER and Robinhood.

Resource Utilization

Table 6.8 shows the effect of varying LRU cache size in re-indexer. The best event reporting rate with an optimal resource utilization is achieved when cache size is set to 5000. Therefore, re-indexer does not utilize a lot of cpu (2.94%) and memory (62.4 MB) and can run continuously to keep track of the metadata events in real-time.

Cache Size (#fid2path)	**CPU% on client**	**Memory (MB) on client**	**Events/sec reported by Brindexer**
200	4.8	88.7	578
500	3.5	84.3	624
1000	2.98	75.6	659
2000	2.95	61.3	698
5000	2.94	62.4	734
7500	2.92	60.7	720

Table 6.8: BRINDEXER performance and resource utilization vs. cache size.

6.5 Chapter Summary

In this chapter, we have presented BRINDEXER, a metadata indexing tool for large-scale HPC storage systems. BRINDEXER has an in-tree design where it uses a parallel leveled partitioning approach to partition the file system namespace into disjoint sub-trees which can be indexed in parallel. BRINDEXER maintains an internal metadata index database which uses a 2-level database sharding technique to increase indexing and querying performance. BRINDEXER also uses a changelog-based approach to keep track of the metadata changes and re-index the file system. BRINDEXER is evaluated on a 4.8 TB Lustre storage system and is compared with state-of-the-art GUFI and Robinhood engines. BRINDEXER improves the indexing performance by 69% and the querying performance by 91% with optimal resource utilization.

In future, we plan to implement the re-indexer within the indexer of Brindexer so that there is no overhead from reading and writing entries to suspect files. We also plan to implement Brindexer for other HPC storage systems like BeeGFS and IBM Spectrum Scale.

Chapter 7

I/O Load Balancing for Big Data HPC Applications

7.1 Introduction

High performance computing (HPC) is routinely employed in many science domains such as Physics, and Geology, to simulate and understand the behavior of complex phenomena. Big data driven scientific simulations are resource intensive and require both computing and I/O capabilities at scale. There is a crucial need for revisiting the HPC I/O subsystem to better optimize for and manage the increased pressure on the underlying storage systems from big data processing.

Several factors affect the I/O performance of big data HPC applications. First, the number and kinds of applications that an HPC storage system supports is increasing rapidly [213], which leads to increased resource contention and creation of hot spots where some data or resources are consumed significantly more than others. Second, the underlying storage systems, e.g., Ceph [201], GlusterFS [46], and Lustre [47], are often distributed, and adopt a hierarchical design comprising thousands of distributed components connected over complex network topologies. Managing and extracting peak performance from such resources is non-trivial. With changing application characteristics, static approaches (e.g., [68, 198]) are no longer sufficient, necessitating dynamic solutions. Third, the storage components can develop load imbalance across the I/O servers, which in turn impacts the performance and time to solution for the big data problem. Consequently, achieving load balancing in the storage system is a key for achieving a sustainable solution.

Load balancing for HPC storage systems is crucial and is being actively studied in recent works [71]. Extant systems typically attempt to perform load balancing by either having limited support for read shedding to redirect read requests to replicas of the primary copy, e.g., in Ceph [135], or performing data migration. Alternatively, per-application load balancing has also been considered to balance the load of an application across the various I/O servers [198]. These existing approaches lack a global view of all the components in the hierarchical structure of the system, and mainly focus on only a small subset of metrics (e.g., only the storage capacity, and not performance of the components). Thus, these approaches cannot guarantee that the aggregate I/O load of multiple big data applications concurrently executing atop a parallel file system (with bursty behavior) is evenly distributed.

Consider the Lustre file system that forms the backend storage in, among other HPC systems, Oak Ridge National Laboratory's (ORNL) Titan supercomputer [43, 72]. The default strategy in Lustre is to allocate storage targets to I/O requests using a round-robin approach. Experiments show that this approach is inclined to either under- or over-utilize the resources due to the bursty nature of applications.

In this chapter, we address the load imbalance problem in Lustre by enabling a global view of the statistics of key components. We select Lustre to showcase our approach as Lustre is deployed on 60 of the top 100 fastest supercomputers [72], and improving its performance will benefit a wide range of applications and users. We go beyond just network load balancing, e.g., as in NRS [166], or per-application approaches, e.g., as in access frequency-based solutions [198], to ensure that the Lustre Object Storage Targets (OSTs) that actually store and serve the data along with other I/O system components are load balanced. We leverage the existing hierarchy of Lustre to avoid introducing additional performance bottlenecks, and co-locate the global component of our load balancer on Lustre's Metadata Server (MDS) that has a global view of all other components.

Our goal is to improve the end-to-end performance of HPC storage systems for big data applications. Our data-driven approach learns system behavior to better manage the load across various Lustre components. Specifically, we make the following contributions.

- We design a model for the Lustre file system to incorporate a load balancing strategy that considers the global view of the system parameters.
- We utilize a scalable publisher-subscriber model to monitor and capture the load of key components in Lustre. We use a Markov chain model that learns and predicts the future behavior of the application using the monitored data, and a minimum-cost maximum-flow algorithm to assign storage targets in a global load aware fashion.
- We design a realistic trace-driven Lustre simulator that captures the load imbalance behavior. We use the simulator and a real setup to study our approach and design decisions therein.
- We also evaluate the effectiveness and scalability of our approach. Experiments show that our approach helps in achieving a better load-balanced storage servers, which in turn can yield improved end-to-end system performance.

7.2 Background and Motivation

In this section, we first present an overview of the existing load balancing used for Lustre. Then we present a quantitative study of the load imbalance in the HPC I/O subsystem to motivate our approach.

The default Lustre implementation uses a round-robin approach coupled with disk utilization measure to balance the load across Object Storage Targets (OSTs). The first available OST is selected to store a file and if the OST still has available space (more than a predefined threshold), is then placed at the end of a list for the next round . This technique does not consider the load on other resources (MDS or Object Storage Servers (OSS)) or the I/O load on OSTs, and can lead to performance degrading hot spots.

Network Request Scheduler (NRS) [166] aims to achieve distributed network load balancing at the Lustre server level by reordering incoming RPCs to a Lustre server (e.g., MDS or OSS) so that individual Metadata Storage Targets (MDTs) or OSTs running on the server receive a fair share of the server's network resources. Our approach differs from NRS in that we go beyond considering only the network resources to include the many factors affecting I/O performance on various Lustre components (Table 7.1), and aim to provide a globally balanced I/O subsystem to offer more stable I/O performance.

7.2.1 Progressive File Layout

Striping in Lustre file system enables users to obtain high I/O performance throughput [98]. Data is divided into stripes according to a striping pattern and the striped data is stored across OSTs in a round-robin fashion. Progressive file layout (PFL) [134] is a recent feature in Lustre where a file can have a series of flexible striping layouts. Using PFL, a file can be created with several non-overlapping extents, with each extent having different striping parameters.

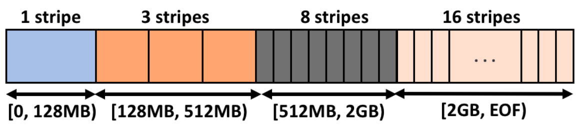


Figure 7.1: An example PFL layout.

An example of a 4-component PFL layout is shown in Figure 7.1. The first component with extents ranging from zero to 128 MB has only one stripe. As the file size increases beyond 128 MB till 512 MB, the file will be divided into three stripes, from 512 MB to 2 GB, the number of stripes is eight, and beyond 2 GB till the end of file, the file will have 16 stripes. The PFL feature is implemented using composite file layouts for regular files. The number of sub-layouts in each file and the number of stripes in each sub-layout can be specified by users

Component	Factors	Discussion

Metadata Server (MDS)	`CPU%`
	`Memory%`
	`/proc/sys/lnet/stats`
		CPU and memory utilization reflect the system load.
		Load on the Lustre networking layer connected to MDS.

Metadata Target (MDT)	`mdt.*.md_stats` `mdt.*.job_stats`	Overall metadata stats per MDT. Metadata stats per MDT per job.

Object Storage Server (OSS)	`CPU%`
	`Memory%`
	`/proc/sys/lnet/stats`
		Reflects the system load of the management server.
		Load on the Lustre networking layer connected to OSS.

Object Storage Target (OST)	`obdfilter.*.stats`
	`obdfilter.*.job_stats`
	`obdfilter.*OST*.kbytesfree`
	`obdfilter.*OST*.brw_stats`
		Overall statistics per OST.
		Statistics per job per OST.
		Available disk space per OST.
		I/O read/write time and sizes per OST.

Table 7.1: List of I/O performance statistics for relevant system components.

using the `lfs setstripe` command. An example command for the PFL layout in Figure 7.1 is given below, where parameters `E` and `c` specify the extent and stripe count respectively.

```
lfs setstripe -E 128M -c 1 -E 512M -c 3 -E 2G -c 8 -E -1 -c 16 <filename>
```

7.2.2 I/O Performance Statistics

Lustre is a hierarchical system. We identify the factors that affect the I/O performance in every layer of the system as shown in Table 7.1. We utilize the files `/proc/meminfo` and `/proc/loadavg` to capture the memory and CPU utilization, and also read the values of Lustre parameters in various components of the hierarchical system, e.g., via `obdfilter.*OST*.brw_stats`.

The performance for serving metadata and the I/O rate of serving the actual data to the clients is dependent on the network performance. A congested network affects the I/O performance adversely. The network bandwidth information can be extracted using the `lnet stat` interface (`/proc/sys/lnet/stats`), which runs over LNET and Lustre network drivers.

7.2.3 The Need for Load Balancing

We conducted a simulator-based study to demonstrate the need for balanced load placement across Lustre components. Our simulator (Section 7.4.1) faithfully implements the functionalities of various components in Lustre. In our model, we have 8 OSSes and every OSS is linked to 4 OSTs (a total of 32 OSTs). We use 24-hour traces from the combination of three application traces to drive the simulator. Two of the application traces are from the Interleaved-Or-Random (IOR) benchmark [121] and the Hardware Accelerated Cosmology Code (HACC) Application I/O kernel [120], while the third application trace is generated from a HPC Transaction Processing Application (TPA) running at a large financial institution [190]. For this test, we use the default round-robin approach of Lustre for allocating OSTs for file creation requests. All results shown are the average taken from three runs.

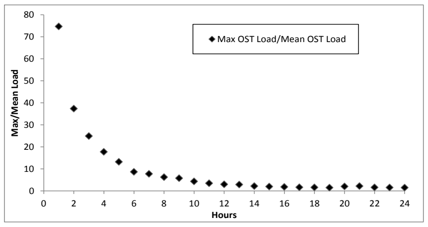


Figure 7.2: Max/Mean OST load over time (Round-Robin).

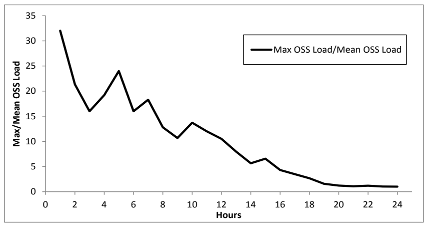


Figure 7.3: Max/Mean OSS load over time (Round-Robin).

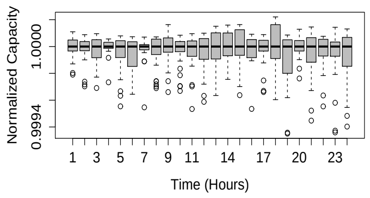


Figure 7.4: Capacity of all OSTs over time (Round-Robin).

We measured load balance by taking the ratio of maximum system load to the mean system load—system load for OSTs is the disk space used, and for OSSes is the CPU utilization. This ratio should be one for ideal load balance. For OSTs, we measure the ratio of maximum disk space consumed taking all OSTs into account to the mean disk space consumed for every hour for 24 hours as shown in Figure 7.2. We see that the system starts from a highly unbalanced setup of OSTs and takes over 12 hours to get close to 1. Thus round-robin is very slow in providing load balance.

In addition to load balancing OSTs, our objective is to also have a load balanced set of OSSes. We study the ratio of maximum CPU utilization taking all OSSes into account to the mean CPU utilization for a period of 24 hours with 1-second interval as shown in Figure 7.3. This ratio should also be ideally 1. As seen in the graph, there are huge fluctuations in the ratio for a very long window of 20 hours, again highlighting the weakness of the round-robin approach.

Since, load on OSTs is a continuous function (not represented in Figure 7.2), we also plot load of OSTs along time in a box-plot. Capacity in an OST is the amount of free disk space available. As the capacity continuously changes every hour, we normalize it to the median OST capacity for that hour. This is shown for the studied 24-hour period in Figure 7.4. The box plot highlights the variation in the capacities of all OSTs combined for different hours. Round-robin policy is unable to balance the system due to lack of consideration for numerous other factors as discussed earlier. As the trace file from IOR benchmark generates a continuous stream of file write requests instead of a more complex bursty pattern, the system still becoming unbalanced shows the weakness of the default approach to capture the application behavior and create a well-balanced and application-attuned load distribution.

This study highlights the need for better load balancing for the HPC I/O subsystem. Moreover, an opaque round-robin approach is incapable of accounting for workload dynamicity [198] such as that created by regular purge of old data in HPC systems (by OLCF

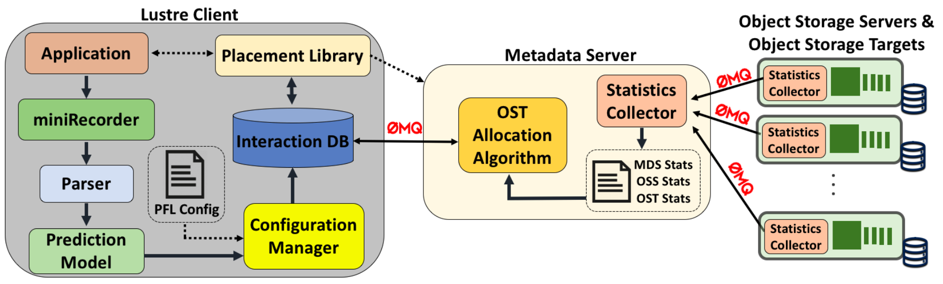


Figure 7.5: Overview of the proposed architecture.

practice [44]). In this chapter, we address such problems by designing an OST management layer that provides a global and detailed system state view to the allocator.

7.3 System Design

We have implemented our load balancing design for the widely-used Lustre file system. However, our design can be extended for use in other HPC parallel file and distributed storage systems that employ a similar hierarchical structure.

Figure 7.5 shows an overview of the architecture. It presents an "end-to-end" control plane for managing I/O with components both on the client side and the server side. When running applications for the first time, the client side makes use of a customized tracing tool, `miniRecorder`. This tool collects information about the I/O accesses, such as the number of bytes written, file name, number of stripes, and MPI rank and communicator for each file. `MiniRecorder` needs to collect traces only for the first run of an application.An application's I/O behavior is identified, which does not change across multiple runs of an application. The collected traces are fed into the `parser` that then uses the information to drive the `prediction model`. Our predictions are based on ARIMA time series modeling [48]. The output of the time series prediction provides estimates of future application requests, which are sent to the `configuration manager`. The configuration manager is responsible for determining if the layout for an application is PFL. `PFL Config` stores all the PFL configurations added by a user which is used by the configuration manager. The output is then stored in an `interaction database` for later use. We refer to the database as "interaction database" because it offers a point of interaction between our server-side and client-side libraries.

On the server side, OSSs collect the CPU and memory usage information, associated OSTs capacity (*kbytestotal*) and the number of bytes available on the OSTs (*kbytesavail*). These statistics are sent to the MDS using the `statistics collector` module on the OSSs. The collected information on the MDS is parsed in the `statistics collector` placed on the MDS to generate a file containing updated statistics for the MDS, OSS and associated OSTs. This file is fed to the `OST allocation algorithm`. The input to the OST allocation algorithm is the predicted set of requests received from the clients from the `interaction database` via ZeroMQ message queues [86], and the output is the list of OSTs to be allocated for every request, which will yield a load-balanced distribution over the involved OSSs and OSTs. The allocated OSTs are stored in the interaction database along with the predicted requests. Next, the `placement library` intercepts the I/O requests from the applications, consults the interaction database, and routes the application requests to appropriate resources by creating a given file's metadata on the MDS.

7.3.1 Parallel File Access Modes for Varying Striping Layouts

Before explaining the different components, we show how different file access modes - File-per-process (FPP) and Single-shared-file (SSF) are represented in PFL and non-PFL layouts. We show an overview of the representation in Figures 7.6, 7.7, 7.8, and 7.9. The PFL layout used in the example is *Configuration 1* discussed in Section 7.2.1.

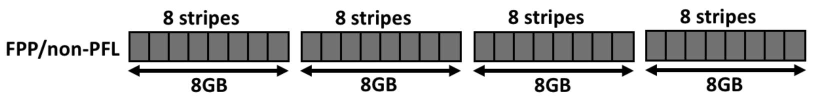


Figure 7.6: Layout in FPP mode - 4 processes each writing 8GB in a non-PFL setup.

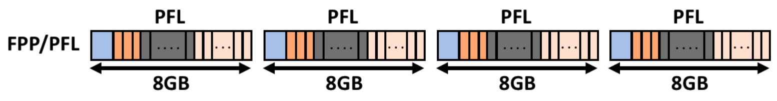


Figure 7.7: Layout in FPP mode - 4 processes each writing 8GB in a PFL setup.

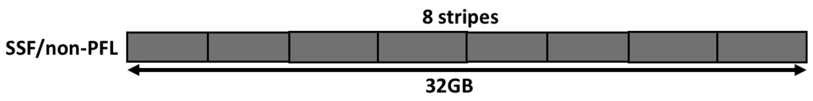


Figure 7.8: Layout in SSF mode - 1 process creating a 32GB file in a non-PFL setup.

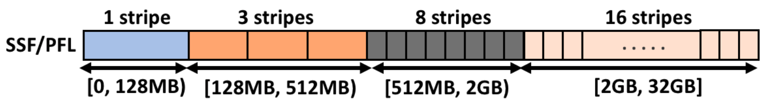


Figure 7.9: Layout in SSF mode - 1 process creating a 32GB file in a PFL setup.

For FPP mode, we have four processes each writing 8 GB to the parallel file system. In the non-PFL layout, each 8 GB file will be striped into a pre-defined number of stripes (8 in the figure), where as, in the PFL layout each 8 GB file will be striped according to the PFL layout set by the user for those files or directories (*Configuration 1*). In SSF mode, a single process creates a single file where all other processes perform I/O operations. This file will be striped into a pre-determined number of stripes according to a non-PFL or a PFL layout. It is evident that different stripes will be of different sizes based on the file access modes as well the striping layouts. The challenge lies in selecting OSTs to place these varying size stripes such that all OSTs are load balanced and have less resource contention, which improves the I/O throughput.

7.3.2 Client-Side Components

In the following, we describe the components that run on the clients.

Tracing Tool

We implement a simple and lightweight I/O tracing library, `miniRecorder`, based on Recorder [127]. Recorder is a multi-level I/O tracing framework, which can capture I/O function calls at multiple levels of the I/O stack, including HDF5, MPI-IO, and POSIX I/O. For our end-to-end system, we limit the range of intercepted function calls to file creation and write calls, and record the bytes written, file name, stripe count, and MPI rank and communicator for each file. We focus on writes more than reads because caching mechanisms and burst buffers that are typical in modern HPC deployments absorb most of the read requests once the file has been written. Therefore, the load imbalance is mainly due to write requests [127, 151]. The traced data is processed by the `parser` and converted into a readable (comma separated) *.csv* format file. This file is then sent to the prediction library (ARIMA Inspired Prediction Algorithm discussed in the next section). The tracing tool is lightweight and our tests show that it adds a negligible memory and CPU overhead of ∼0.3% and <1%, respectively, during application execution.

ARIMA-Inspired Prediction Algorithm

HPC applications have been known to show distinct I/O patterns [151]. Based on our interactions with HPC practitioners, this predictability is expected for emerging applications as well. We leverage this observation to model three key properties of HPC I/O, namely, write bytes, stripe count, and MPI rank. We collect these parameters using the tracing tool on the first run of an application to train our model. Previous works [19, 38, 39] present auto-tuning approaches for MPI-IO and Lustre to learn and predict the I/O parameters for improving read and write performance of HPC applications. These approaches use a range for Lustre stripe-counts, I/O buffer sizes and I/O bandwidth of previous runs. We use AutoRegressive Integrated Moving Average (ARIMA) model [48] to fit our multivariate time series data and predict future values. Our choice of ARIMA is dictated by its performance and the time-series nature of the write bytes and stripe count of the requests. We experimented with several alternatives, such as the Markov Chain Model [170], to model the data. However, ARIMA yielded better accuracy with lower memory and computing overhead. For example, we observed a 99.1% accuracy in IOR data using ARIMA, while Markov chain model yielded an accuracy of 95.5%. The CPU overhead for ARIMA was less than 1.2% compared to 4.5% in Markov Chain, while the memory usage for ARIMA is 10 MB in comparison to 90 MB usage in Markov chain.

We implement our prediction model on the client side, where the calls would be intercepted, rather than on MDS. This has two advantages: i) since each application has its own client, the model can be applied at scale without overwhelming the centralized MDS; and (ii) the approach makes the MDS application-agnostic, where the server can focus on write requests and types in a global fashion and not be concerned with individual applications. The approach also provides for a much more efficient solution when multiple applications run on (multiple) clients simultaneously.

A time series is defined as a sequential set of data points, measured typically over successive times. It is represented as a set of vectors $x(t)$, $t = 0, 1, 2...$, where t is the elapsed time [45]. The term `ARIMA` involves three parts, `AR` denotes that the variable is regressed on its prior values, `I` stands for 'integrated', which means that the data values are replaced with the difference between the present and previous values, and `MA` represents the fact that the regression error is a linear combination of error terms occurring in the past. There are three parameters used for every ARIMA model. The parameter `'p'` is the number of lag observations (lag order), `'d'` denotes the number of times raw observations are differenced (degree of differencing), and `'q'` represents the size of moving average window (order of moving average).

The first step is to select the values of parameters `(p, d, q)`. To this end, we run the model on all combinations (skipping the ones that fail to converge) of the parameters over our dataset, which we get from the tracing tool, and select the combination with the least Root Mean Square Error (RMSE). We vary the values of p, d, and q from 0 to 5. We go from 0 to 5 for all the values because going beyond 5 would be computationally expensive. For *HACC-IO*, the least RMSE was found for `(5, 1, 2)` and *IOR* gave the minimum RMSE for `(2, 1, 1)`. The next step is to fit the ARIMA`(p, q, d)` model by exact maximum likelihood via Kalman filters [84]. This fitted model is then used to predict the write bytes, stripe count, and MPI rank values for future application I/Os. We use `statsmodels.tsa.arima_model` package in Python for our ARIMA implementation. Our results show a 98.3% accuracy in HACC-I/O data and 99.1% accuracy in IOR data.

Configuration Manager

The `configuration manager` determines if an application is performing I/O in PFL or non-PFL layout. It takes as input the predicted set of requests, containing the file name, stripe count, write bytes, and MPI rank, which is given by the `prediction model`, and the file

File Name	File Size	Extent ID	Extent Start	Extent End	Stripe Size	Stripe Count	MPI Rank
/mnt/lustre/ior/test.0	8589934592	1	0	134217728	134217728	1	0
/mnt/lustre/ior/test.0	8589934592	2	134217728	536870912	134217728	3	0
/mnt/lustre/ior/test.0	8589934592	3	536870912	2147483648	201326592	8	0
/mnt/lustre/ior/test.0	8589934592	4	2147483648	8589934592	402653184	16	0
/mnt/lustre/ior/test.1	8589934592	1	0	134217728	134217728	1	1
/mnt/lustre/ior/test.1	8589934592	2	134217728	536870912	134217728	3	1
/mnt/lustre/ior/test.1	8589934592	3	536870912	2147483648	201326592	8	1
/mnt/lustre/ior/test.1	8589934592	4	2147483648	8589934592	402653184	16	1

Table 7.2: Interaction database snapshot for IOR in FPP mode in PFL layout.

containing all PFL layouts for a client. If the file names or the application directory in the predicted requests set match those in the PFL configuration file, the stripe size and stripe counts for the file are set based on the PFL configuration. If there are no matches found for the file name or the directory in the PFL configuration, then the layout is non-PFL. The output of the configuration manager is the set of all predicted requests combined with the corresponding stripe size, stripe count, and MPI rank of the files. This entire set is sent to the `interaction database`. The stripe size for both layouts is calculated for all files in the configuration manager based on the 64k-alignment constraint by Lustre, which is explained below.

Lustre's 64k-alignment constraint for stripe size Stripe size is an important parameter for load balancing in a distributed file system. Every stripe needs to be assigned to an OST. Intuitively, in order to calculate stripe size, we can divide the total file size by the stripe count. Both of these parameters are given as part of the input requests set in the *configuration manager*. But calculating stripe size is non-trivial because Lustre imposes the constraint that in order to place stripes into the allocated OSTs, stripe size should be even multiples of 64k. We term 64k or 65536 bytes as Alignment Parameter (AP). This constraint becomes a problem for files which are not AP aligned, for example when the total file size is 803405824 bytes, which is not an even multiple of AP. We consider two ways to overcome this constraint.

Method-I: This method assumes that we can allocate equal number of even multiples of AP into the first *(stripeCount - 1)* number of OSTs, and the remaining even multiple of AP goes into the last OST. Equation 7.1 gives the total allocation such that the stripes are AP aligned. The first part of the equation is the placement on first *(stripeCount - 1)* number of OSTs and the second part is the number of bytes written on the last OST.

$$\begin{aligned} writeBytes = {} & (AP * 2 * N * (stripeCount - 1)) + \\ & (AP * 2 * X) \end{aligned} \tag{7.1}$$

where,

$$N = \left\lfloor \frac{writeBytes}{AP * 2 * (stripeCount - 1)} \right\rfloor \tag{7.2}$$

The remaining number of bytes to be written on the last OST is then given by:

$$remainingBytes = writeBytes - (AP * 2 * (stripeCount - 1)) \tag{7.3}$$

Therefore,

$$X = \left\lceil \frac{remainingBytes}{AP * 2} \right\rceil \tag{7.4}$$

Note that we round down the allocation in the first *(stripeCount - 1)* number of OSTs and round up the allocation in the last OST. Multiplication with 2 ensures that the stripe size is an even multiple of AP. Stripe size for the last OST is $(AP * 2 * X)$, and for each of the remaining OSTs is $(AP * 2 * N)$. However, a major drawback here is that this method does not place the load evenly among all the OSTs. The number of bytes written on the last OST will always be smaller compared to the bytes written on the other OSTs for a particular file.

Method-II: This method overcomes the drawback of the previous method by allocating even multiple of AP in all the OSTs.

$$writeBytes = AP * 2 * N * stripeCount \tag{7.5}$$

where,

$$N = \left\lceil \frac{writeBytes}{AP * 2 * stripeCount} \right\rceil \tag{7.6}$$

Stripe size for all the OSTs is given by $(AP*2*N)$. Both methods ensure an even multiple of 64k-alignment of stripe size for all the stripes allocated in the *stripeCount* number of OSTs, by allocating a slightly bigger file than was requested by the client. The second method places all the stripes equally on all the OSTs but needs a bigger file to be allocated in comparison to the first method. Thus, there is a trade-off between how big the file allocation we can allow versus balancing all the stripes among the OSTs. Our results show that for a 766.175 MB file size, we allocated 833 KB (0.1%) bigger file using the second method and 63 KB (0.008%) more in the first method. We proceed with the second method because in spite of allocating a little more than was requested by the client, this approach ensures allocating equal stripes on all the OSTs. This would lead to similar load accesses from all OSTs and OSSs, therefore approaching towards a load-balanced setup.

Interaction Database

The `interaction database` is a SQL database located on the Lustre clients. It serves as the medium through which the MDS and clients interact with one another. First, the output set from the `configuration manager` is stored in the database. Different tables are used to store PFL and non-PFL layout files. Tables 7.2 and 7.3 show an example snapshot of the interaction database for IOR in FPP mode with 2 processes each writing 8 GB in PFL and non-PFL layout respectively. For PFL layout, we use the example PFL configuration (*Configuration 1*), discussed in Section 7.2.1. As seen in Table 7.2, every file is associated with all the extents specified in the PFL configuration. For each file, we store the file name, the file size in bytes, the extent ID from the PFL configuration file, the corresponding start and end range for the extent, stripe size, stripe count and the MPI rank. For non-PFL layout, shown in Table 7.3, we store the file names, stripe size of the files, number of stripes associated with every file, and the MPI Rank. The stripe size of the files is calculated using the 64k-alignment parameter which is discussed in Section 7.3.2.

File Name	**Stripe Size**	**Stripe Count**	**MPI Rank**
/mnt/lustre/ior/test.0	1073741824	8	0
/mnt/lustre/ior/test.1	1073741824	8	1

Table 7.3: Interaction database snapshot for IOR in FPP mode in non-PFL layout.

The MDS uses a scalable publisher-subscriber model via Zero message queue [86] to retrieve the required contents from the interaction database. The pub-sub model helps in scaling to a large number of clients [152]. The specific steps are discussed in Section 7.3.3. For our implementation, we use MySQL 8.0.12 Community Server Edition. Our results show that writing and retrieving data from the interaction database is very efficient, using <0.3% and <0.4% of CPU and memory, respectively.

Placement Library

The placement library complements the prediction model by providing a lightweight, portable, and user-friendly mechanism to access and apply the predicted file layout to an application's I/O workload (i.e., without any code modification). the placement library relies on function interpositioning provided by the `GNU` dynamic linker to prioritize itself over standard system calls, and the profiling interface to MPI (PMPI). Hence, it can be used by setting the environment variable `LD_PRELOAD` to the path of the shared library. Metadata operations (e.g., `open()`) issued by the application are intercepted and redirected to the placement library. Following the example layouts shown in Figures 7.6, 7.7, 7.8, and 7.9, the placement library supports both non-PFL and PFL layouts for FPP and SSF.

For every I/O cycle, the placement library queries the interaction database via the MySQL C API with the file name passed by the original metadata operation, fetches the matching rows, and applies the predicted striping pattern to the file, if the file does not exist yet. To facilitate this, Lustre provides a user library called *llapi*, which allows the user to describe a specific striping pattern. However, the placement library cannot use *llapi* directly, since it internally triggers `open()` calls, which would result in a continuous, recursive loop due to nature of the preloading mechanism. Therefore, the placement library mimics the behavior of *llapi* and communicates directly with the Lustre Logical Object Volume (LOV) client to create the file metadata on the MDS, similarly to [137, 138, 196].

If the result returned by the MySQL query contains only one row, the non-PFL layout is applied by allocating a Lustre file identifier and the corresponding *Layout Extended Attributes* (Layout EA) on the MDS. For both FPP and SSF, the predicted striping pattern is applied by initializing the Layout EA with the stripe count, stripe size, and list of OSTs retrieved from the interaction database.

If the MySQL query returns more than one row, the PFL layout is utilized. For both FPP and SSF, every row represents one non-overlapping extent for a given file. the placement library iterates over all rows, allocates an array of sub-layout components (one for each file extent), and applies the predicted striping pattern to the file by storing it in a composite layout on the MDS. Composite layouts, unlike Layout EA, allow the specification of different specific striping patterns for different ranges (i.e., extents) in the same file.

Algorithm 8 presents a simplified overview of the placement library, which is run on every client. It currently supports POSIX I/O, MPI-IO, and HDF5 and the following I/O calls: `open[64]()`, `creat[64]()`, `MPI_File_open()`, and `H5Fcreate()`. The key advantage of this transparent approach is that applications can directly benefit from the prediction model without modifying the source code.

Algorithm 8: File layout creation on the MDS.

```
Input: File Name $file$, Access mode $flags$
Output: Call to real metadata operation (e.g., $open()$)
begin
    if fileExits(file) == TRUE then                     // File exits; return.
        return realMetadataOperation(file, flags)
    $flags = flags$ | O_LOV_DELAY_CREATE
    $result$ = queryInteractionDatabase($file$)
    if numMySQLrows(result) == 1 then                   // Non-PFL layout.
        $row$ = fetchMySQLrow(result)
        $layoutEA$ = allocLayoutEA($row.stripeCount$, $row.stripeSize$, $row \rightarrow OSTs$)
        createLayoutEAonMDS($file$, $flags$, 0644, $layoutEA$)
    else if numMySQLrows(result) > 1 then               // PFL layout.
        while row = fetchMySQLrow(result) do
            allocExtentPFL($layoutPFL$, $row.extentEnd$, $row.stripeCount$,
              $row.stripeSize$, $row \rightarrow OSTs$)
        createCompositeLayoutMDS($file$, $flags$, 0644, $layoutPFL$)
    return realMetadataOperation(file, flags)
```

7.3.3 Server-Side Components

In the following, we describe the components that run on the servers (OSSs and MDS) and how they interact with each other.

Statistics Collector on OSS

Statistics collection is done for every OSS. The list of all the OSTs for a particular OSS is saved in a configuration file that is provided as input to the `statistics collector` for that specific OSS along with the OSS ID. For every OST, we collect the total and available capacity, found in the files $/proc/fs/lustre/obdfilter/ost\_name/kbytestotal$ and $/proc/fs/lustre/obdfilter/ost\_name/kbytesavail$, respectively. We also collect the CPU and memory utilization of the OSS by reading data from the files $/proc/meminfo$ and $/proc/loadavg$. The statistics collection algorithm runs every 60 seconds on all the OSSs. We choose 60 seconds as our interval for statistics collection so that we get updated statistics on the MDS without over-loading the OSSs. These statistics are sent to the MDS.

The load monitoring (statistics collection) solution needs to be scalable. Therefore, we use a publisher-subscriber model [152] for the statistics collection framework. OSSs act as publishers and MDS as the subscriber. Statistics collected in the OSSs are sent to the MDS via a message queue. We use ZeroMQ (ϕMQ) [86] as our message queue because it is lightweight and has been shown to be very efficient at large scale [151, 152, 153, 196]. We also collect the CPU and memory utilization of the MDS every 60 seconds, in the same way as it is collected in the OSSs. Our tests with the implementation show that the statistics collection framework on average has negligible CPU and 0.1% memory utilization on the OSSs, and 0.6% CPU and 0.1% memory on the MDS.

Statistics Collector on MDS

The `statistics collector` on the MDS is responsible for the following:

- Collecting statistics from the MDS.
- Subscribing to statistics from the OSSs via ZeroMQ.
- Parsing all the collected statistics.

The CPU and memory utilization collected in the MDS is important to determine when the OST allocation algorithm will run. The OST allocation algorithm runs only if the CPU utilization is lower than 70% and memory utilization is lower than 50%. This is done so that the load balancing algorithm does not disrupt the normal functionalities of Lustre's MDS. The collected statistics from the MDS and OSSs are parsed and is sent as input to the OST

File Name	File Size	EID	Extent Start	Extent End	Stripe Size	#Stripe	MPI Rank	OST List
test.0	8589934592	1	0	134217728	134217728	1	0	10
test.0	8589934592	2	134217728	536870912	134217728	3	0	24 29 34
test.0	8589934592	3	536870912	2147483648	201326592	8	0	15 2 28 25 11 21 7 33
test.0	8589934592	4	2147483648	8589934592	402653184	16	0	14 23 16 30 1 6 4 26 ...
test.1	8589934592	1	0	134217728	134217728	1	1	24
test.1	8589934592	2	134217728	536870912	134217728	3	1	13 9 29
test.1	8589934592	3	536870912	2147483648	201326592	8	1	22 11 35 21 5 7 17 10
test.1	8589934592	4	2147483648	8589934592	402653184	16	1	14 23 16 30 1 6 15 4 ...

Table 7.4: Interaction database snapshot showing OST allocation for IOR in FPP mode in PFL layout.

File Name	Stripe Size	Stripe Count	MPI Rank	OST List
/mnt/lustre/ior/test.0	1073741824	8	0	30 20 5 1 22 35 14 23
/mnt/lustre/ior/test.1	1073741824	8	1	20 19 1 9 29 2 33 18

Table 7.5: Interaction database snapshot showing OST allocation for IOR in FPP mode in non-PFL layout.

allocation algorithm (Section 7.3.3). Our results show that the statistics collector on the MDS has a CPU utilization of 0.1% and negligible memory utilization.

OST Allocation Algorithm

The input to the `OST allocation algorithm` is the parsed OSS and OST statistics, and the write requests (file name, stripe size and stripe count) sent by the clients.

Algorithm 9: Obtaining list of OSTs for each request.

```
Input: OSS statistics $cpu$ & $mem$, OST statistics $totalKbytes$ & $kbytesAvail$, Write Requests $stripeSize$ & $stripeCount$
Output: $OSTAllocationList$
begin
    for $OSS\ oss\ in\ OSSList$ do
        $ossLoad = (cpuweight * cpu) + (memweight * mem)$
        for $OST\ ost\ in\ OSTList$ do
            $ostCostToReach = OSSLoad$
            $ostCost = (totalKbytes - kbytesAvail)/totalKbytes$
            $ostCapacity = kbytesAvail/stripeSize$
    flowGraph = buildGraph(Requests, OSS, OST)
    OSTAllocationList = minCostMaxFlow(flowGraph)
    return ($OSTAllocationList$)
Function buildGraph
    Input: Requests $req$, StripeCount $sc$, OSTCostToReach $ossLoad$, ostCost $ostLoad$, OSTCapacity $ostCap$
    Output: FlowGraph $G$
    totalDemand = sum of stripeCount for all Requests
    G.addNode('source', totalDemand)
    G.addNode('sink', -totalDemand)
    for $request\ r\ in\ reg$ do
        G.addEdge('source', $r$, cost = 0, capacity = sc)
        for $OST\ ost\ in\ ostList$ do
            G.addEdge($r$, $ost$, cost = ossLoad, capacity = 1)
    for $OST\ ost\ in\ ostList$ do
        G.addEdge($ost$, 'sink', cost = ostLoad, capacity = ostCap)
    return ($G$)
```

Algorithm 9 shows the the OST allocation algorithm, which employs a minimum-cost maximum-flow approach [20]. The flow graph that is used to solve the problem is shown in Figure 7.10. We calculate the cost to reach an OST (which is the load of the OSS), the cost of an OST (ratio of bytes already used in the OST to the total size of the OST), and the capacity of an OST (the number of stripes that can be handled by the OST, given by the ratio of available space in the OST to the stripe size). In order to derive the flow graph, we need to identify the *source* and *sink* nodes. The total demand for the *source* node is the total number of stripes requested, and the total demand for the *sink* node is the negative amount of the total number of stripes requested. We solve the minimum-cost maximum-flow using the Ford-Fulkerson algorithm [188]. This outputs a list of OSTs ($OSTAllocationList$) using which will yield a balanced load over both OSS and OSTs. For our implementation, we use the `networkx` library in Python. Our results show that the algorithm on average uses 1.58% CPU and 0.1% memory on the MDS.

The list of OSTs obtained from the OST allocation algorithm are then sent to the respective clients using the publisher-subscriber model via ZeroMQ. The complete set of requests are stored in the interaction database. Example entries for the database with the complete allocation for an IOR application in FPP mode with 2 processes each writing an 8 GB file in both PFL (*Configuration 1*) and non-PFL layouts are shown in Tables 7.4 and 7.5 respectively. We add a new column *OST List* in the database. The *OST List* is a space separated load-balanced list of OSTs for every write request. This example is for a setup with 7 OSSs and 35 OSTs (5 OSTs associated with every OSS) – therefore, OST ids range from 1 to 35. As described earlier, the placement library then uses this information to place the requests, thus completing the load-balanced allocation of resources. If for any run of the application, the placement library is unable to find more than 50% of files in the interaction database, `miniRecorder`, ARIMA prediction and OST Allocation Algorithm will be executed again to update the interaction database.

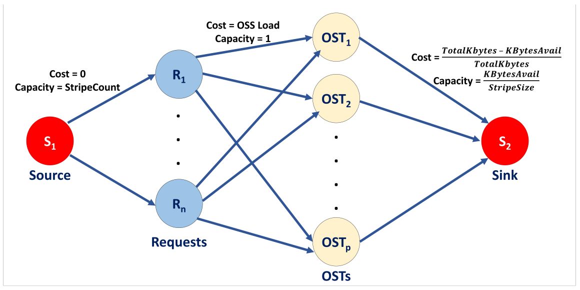


Figure 7.10: Graph used in OST Allocation Algorithm.

Summary: Components run both on the client and server side of a Lustre deployment. `MiniRecorder` runs on the clients during the first execution of an application (or if the system cannot find prediction information for most of the accessed files) to capture its I/O characteristics. Next, the traces are fed into the `prediction model` and the predicted set of requests are provided to the `configuration manager` to check the file layout and calculate the stripe size of the files. The client and server libraries interact using the *interaction database*. *Statistics collection* on the OSS and MDS happens periodically. Whenever, the interaction database is updated, it sends the new values to the MDS, which (if not overloaded) executes the `OST allocation algorithm`. The results are sent back to the interaction database. For subsequent application runs, the `placement library` intercepts the application write/create calls, reads from the interaction database and writes the request to a load-balanced set of OSTs.

7.4 Evaluation

We evaluate the efficacy of our approach using both a Lustre simulator and a live setup. In the following, we first describe our simulator and experimentation methodology, then compare our MCMF-based load balancing with the default Lustre OST allocation approach.

7.4.1 Methodology

Simulator

We have developed a discrete-time simulator based on the overall system design shown in Figure 7.5 to test our approach at scale. The simulator has four key components closely mirroring those of Lustre's OST, OSS, MDT, and MDS, which implement the various Lustre operations and enable us to collect data about the system behavior. The MDS is also equipped with multiple strategies for OST selection, such as round-robin, random, and MCMF. We have implemented a wrapper component that enables communication between our various simulator components. The wrapper is responsible for processing the input, managing the MDS, OST, and OSS communication and data exchange, and driving the simulation. All the network components in the simulator are modeled using Network Simulator (NS-3) [140]. The application traces collected from client side are modeled as clients in the simulator. In our simulations, all initial conditions are the same at the start of any allocation strategy. The parameters, number of OSSes, number of OSTs under each OSS are provided as inputs to the simulator.

Cluster Setup

We also conducted experiments on a small Lustre 2.8.0 setup to determine how the various components interact. The client node has 8 cores, 2.5 GHz Intel processor, 64 GB memory, and 500 GB HDD. MDS and two OSSes have 32 cores, 2.0 GHz AMD processors, 64 GB memory, and 1 TB HDD. All components are connected through a 10 Gbps Ethernet interconnect. We have set 6 OSTs in each OSS, each with 170 GB available disk space and

1 MDT in the MDS with a disk space of 100 GB. For real setup test, we repeated each experiment three times, and report the average results.

Workloads

To drive our simulations, we collect application traces at the client side. These application traces contain two kinds of data: (a) write entries, which have the timestamp, the number of bytes to be written, and the number of OSTs to be selected (i.e. the stripe count); and (b) read entry, which has the timestamp, number of bytes to be read and the OST ID from which the bytes have to be read.

To model the behavior of a real Lustre deployment, we run and capture a trace of 3 simultaneously running big data hpc applications on a production Lustre deployment. We use the HACC I/O kernel [120] that measures the I/O performance of the system for the simulation of Hardware Accelerated Cosmology Code (HACC) generating around 12000 file events per second, and the IOR benchmark [121] that is used for testing the performance of parallel file systems which generates around 20000 events per second. The third trace is generated from a high performance computing transaction processing application running at a large financial institution [190]. This trace generates about 15000 file events per second. For our tests (except the scalability study), we simulate the behavior of one MDS, eight OSSes with four OSTs per OSS, for a total of 32 OSTs. We also use our real-setup test to verify the results from the simulator.

7.4.2 Comparison of Load Balancing Approaches

We compare our MCMF based OST load balancing with the standard Lustre round-robin approach, as well as weighted random allocation where OSTs are selected at random from a subset of OSTs. Our random allocation model picks a random OST from a subset of the OSTs whose ratio of available space to total disk space is greater than 0.4. The goal is to remove the OSTs with less available space from the eligible list of OSTs to serve the request.

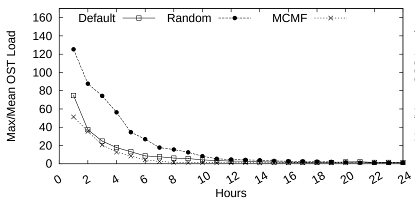


Figure 7.11: Max/Mean OST load over time in simulated setup.

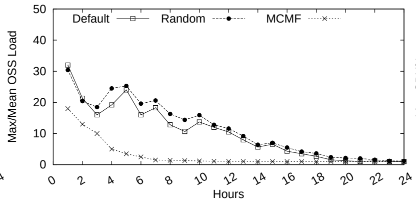


Figure 7.12: Max/Mean OSS load over time in simulated setup.

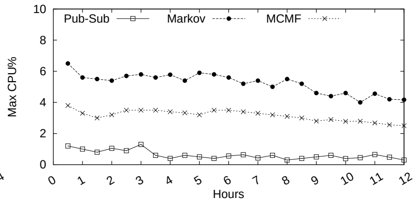


Figure 7.13: Max CPU% on MDS over 12 hours in real setup.

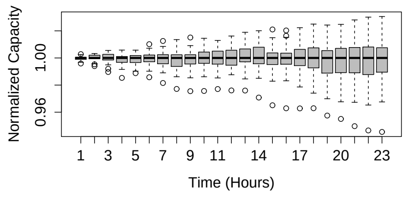


Figure 7.14: Capacity of all OSTs over time under MCMF in simulated setup.

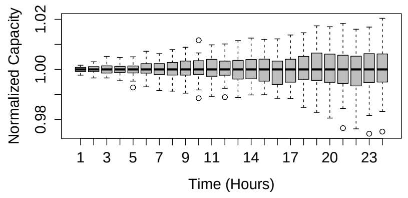


Figure 7.15: Capacity of all OSTs over time under MCMF in real setup.

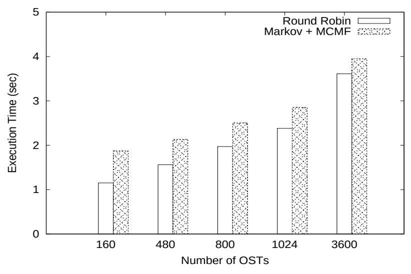


Figure 7.16: Execution time with increasing number of OSTs in simulated setup.

We measure the load balancing in our setup by plotting the ratio of the maximum disk space used to the mean disk space used over time. The ratio should tend to 1 for a load balanced setup. As seen from Figure 7.11, over a period of 24 hours, the random allocation gives the worst result as the ratio starts from 125, which is much more than the starting ratio in round-robin allocation. In contrast, we get better result with the MCMF allocation scheme. The ratio starts from 52 and has a steady decline to 1 in only 7 hours compared to 12 hours in round robin allocation.

In addition to balancing the load across OSTs, our objective is to also load balance the OSSes' CPU utilization—which is ignored under extant round-robin. Figure 7.12 shows the max/mean CPU utilization ratio of OSSes over a period of 24 hours. We see a better load balance in MCMF compared to round-robin and random allocations with a smoother decline of the ratio to 1 in 7 hours compared to 20 hours under round-robin strategy.

Our objective is to load balance OSTs such that every OST is at an almost similar state (in terms of number of bytes available, and load) under various file allocations. Since capacity of OSTs continuously decreases over time, we use normalized capacity where for every hour, the capacities of all OSTs are divided by the median capacity. Figure 7.14 shows the box-plot for normalized capacity vs. time for 23 hours under our approach. When comparing this with the round robin approach (Figure 7.4), we see that the inter-quartile ranges for MCMF are less wider than those for round-robin allocation. This shows that at any given time, MCMF is better able to balance load across OSTs.

We repeat the test on the real setup, and see a similar trend of Max/Mean load on both OSSes (CPU utilization) and OSTs (disk usage) as shown in Figures 7.11 and 7.12. Figure 7.15 shows the normalized capacity over time. The difference in the interquartile ranges in the box plots and a fewer number of outliers is due to the fact that on the real setup, we run 2 applications; IOR and HACC I/O kernel, compared to 3 applications being run on the simulator. The experiment on the real setup also shows that our approach gives better results on the real setup as well.

On the real setup, we also track on the MDS the CPU and memory utilization of our system components, i.e., the publisher-subscriber model, Markov model, and the minimum-cost maximum-flow algorithm. Figure 7.13 shows the maximum CPU utilization on MDS for a period of 12 hours over intervals of 30 minutes for our three components. The maximum CPU utilization is 1.3%, 3.8% and 6.5%, and the maximum memory utilization (not shown in a graph) on MDS are 15.2 MB, 75 MB and 200 MB for publisher-subscriber model, MCMF algorithm, and Markov model, respectively. On OSS, the CPU and memory utilization for the publisher never exceeds 1% and 12 MB, respectively. This shows that our approach requires negligible resources and can easily coexist with Lustre at scale. The Markov model component has the highest CPU and memory utilization, and to keep that in check, we designed the system to train the model and run MCMF algorithm only when the Lustre CPU utilization on MDS goes below a preset threshold (70% in our tests) to avoid any impact on data path performance.

7.4.3 Scalability Study

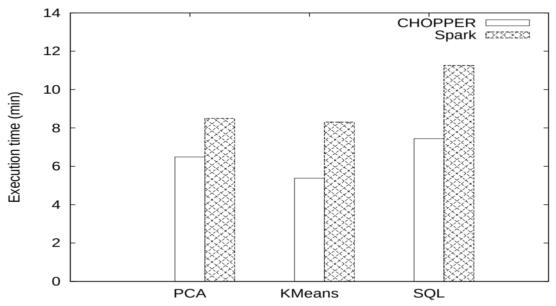


Figure 7.17: Performance with increasing number of OSTs in simulated setup: 160 OSTs (20 OSS), 480 OSTs (60 OSS), 800 OSTs (100 OSS), 1024 OSTs (128 OSS) and 3600 OSTs (450 OSS).

In our next experiment, we test how our approach will work with higher number of storage targets. For this purpose, we use a setup with an increasing number of OSTs from 160 to 3600.

In our simulations, in order to calculate the I/O bandwidth, all OSTs are assigned the same bandwidth at the start. The simulator also takes into account the number of applications using a particular OST at a given timestamp and calculates the read and write bandwidth accordingly. We assume that OSTs have equal read and write bandwidth. Figure 7.17 shows the overall mean I/O bandwidth for Lustre's default round-robin approach as well as our MCMF algorithm. As the number of OSTs increases, the mean bandwidth also increases for both the algorithms. Our algorithm provides better performance than round-robin solution even for higher number of OSTs. As seen from the figure, MCMF allocation is able to achieve up to 54.5% (under 800 OSTs) performance improvement compared to the default round-robin approach.

The overall execution time for load balancing using both round-robin approach as well as our algorithm (Markov model along with MCMF) is shown in Figure 7.16. Round-robin takes less time than our approach to allocate OSTs to incoming requests, but the difference between both execution times keeps on decreasing as we increase the number of OSTs. This is because, the increase in execution time with increase in the number of OSTs is much higher in Round-robin than MCMF. This shows that our approach is more scalable. The gain in the overall performance as shown in Figure 7.17 is much higher than the gain in execution time, even for fewer number of OSTs.

The tests show that MCMF algorithm provides better load balanced allocation of OSTs with improved performance compared to Lustre's default round-robin allocation. Also, the performance of our algorithm does not degrade even with very large number of OSTs. Moreover, with higher number of OSTs, the execution time for our approach to allocate OSTs to requests is similar to that of Lustre's default round-robin approach. This is seen in Figure 7.17, where even for higher number of OSTs, MCMF performs better than the standard approach. Note that the difference in the CPU utilization on MDS when using the default round-robin allocation compared to when MCMF algorithm along with Markov chain model was being executed never exceeds 7.3%. Therefore, the benefits achieved by MCMF algorithm over standard round-robin allocation is achieved at a manageable cost, which is further amortized by the overall application I/O improvement resulting from the better load balanced setup.

7.5 Chapter Summary

We have presented a load balancing approach for extreme-scale distributed storage systems, such as Lustre, where we enable the system to have a global view of the hierarchical structure and thus make more informed and load-balanced resource allocation decisions. We design a global mapper to be located in MDS of Lustre, which uses a publisher-subscriber model to collect runtime statistics of the various components in the I/O system by piggybacking the data on existing communication, employs Markov chain model to predict future application requests based on past behavior, and a minimum-cost maximum-flow algorithm to select OSTs in a load-balanced fashion. Experiments show that our approach provides a better load balanced solution for both OSSes and OSTs than the extant round-robin approach used in Lustre. This will lead to better end-to-end performance for HPC big data applications. In our future work, we plan to incorporate our solution into other systems such as Ceph and GlusterFS, as well as study our approach under different failure scenarios.

Chapter 8

Conclusion and Future Work

A key goal of this dissertation is to build an application-attuned framework for HPC storage systems that will mitigate real-world data management problems. In this dissertation, we have successfully applied the application-awareness principle and methodology to uncover critical issues in distributed and data-intensive systems. Investigations have led to the design and development of effective and novel approaches to solve these problems. For example, to the best of our knowledge, FSMonitor [153] - a scalable file system monitor for large scale storage systems is the first monitor that can be scaled and used on arbitrary storage systems.

This dissertation is driven by the complexities of modern high performance computing and data-intensive systems, and the need for more efficient and flexible approaches to manage such complexities. The work of this dissertation targets three real-world HPC storage systems: *Apache Spark [211]* - an in-memory data analytics framework, *Lustre [47]* - a hugely popular parallel file system, and *Ceph [201]* - an open-source and widely-used object storage system. We use real-work applications - *SparkBench [112]*, Hacc-IO [120], FSMark [8], MDTest [13], and IOR [121]. By performing extensive and deep analysis to understand the issues [157], designing optimization framework [163], practical tools for monitoring [153, 156] and indexing [159] to efficiently make use of complex application behaviors, and building "end-to-end" efficient systems [151, 158, 196] to manage different tradeoffs and the massive volume of data, this dissertation demonstrates improved efficiency and usability of HPC storage systems at the system level with a broad focus on practical and user-centric metrics.

8.1 Summary

High performance computing (HPC) storage systems are increasingly becoming important. According to a recent report from National Energy Research Scientific Computing Center (NERSC) [14], over the past 10 years, the total volume of data stored at NERSC has grown at an annual rate of 30 percent. This massive rate of data generation has resulted in an increasing need for high performance distributed storage systems like Apache Spark [211, 212], Ceph [201] and Lustre [47]. The dependency on HPC storage systems has resulted in input-output (I/O) operations becoming the bottleneck for application performance. Massively parallel HPC applications can suffer from imbalance in computation and I/O performance, with I/O operations becoming a limiting factor in application efficiency [105]. *To mitigate this problem, this dissertation using a holistic redesign approach that cohesively combines piece-by-piece optimizations provides an effort to implement application-attuned framework for high performance storage systems to support the I/O needs of HPC applications.*

We first understand the I/O behavior of HPC applications which is very important for system administrators, file system developers, and HPC users. We collected Lustre file system server level statistics from two clusters, Cab and Quartz at Lawrence Livermore National Laboratory, for a period of three years and analyzed the statistics in an application-agnostic manner. Our studies have indicated interesting results which show that most jobs are write-intensive, showing the importance of improving file system write performance. Our analysis also led us to believe that focus should be on jobs which run for short duration as the majority of the jobs run for less than an hour. Also, there should be efforts to educate HPC users to develop applications which perform efficient writes. This would improve I/O performance as well as help in reducing I/O contention among jobs. We believe that our analysis will help all HPC practitioners to build better file systems and utilize it more effectively.

Second, we design a dynamic partitioning approach for in-memory data analytic platforms by determining the optimal number of partitions and the partitioner for each stage of a running workload with the goal of minimizing the stage execution time and shuffle traffic. We also consider the dependencies between stages, including join and co-group operations, to further reduce shuffle traffic. By minimizing the stage execution time and shuffle traffic, the design implicitly alleviates the task data skew using different partitioners and improves the task resource utilization through optimal number of partitions. Experimental results demonstrate that Chopper effectively improves overall performance by up to 35.2% for representative workloads compared to standard vanilla Spark.

This dissertation then tackles the issue of trying to monitor HPC storage systems by building a generic and scalable file system monitor for capturing and reporting events on heterogeneous large-scale storage systems. The design uses a three-layer approach to file system event monitoring. It implements a standard event definition process for any file system, and works seamlessly for both parallel file systems and object stores. We evaluated our approach on three Lustre file system testbeds and a 8 TB Ceph store. We found that on a 897 TB Lustre system, it reported almost 38 000 events per second with low resource utilization. On the 8 TB Ceph system, `FSMonitor` reported more than 2000 events per second. Compared to iterative monitoring methods used by the popular Robinhood system, our design achieves a 14.5% improved event reporting rate for multiple Lustre MDSs. It also performs well for metadata benchmarks. We also did not notice any performance degradation in data benchmarks when using our monitor. Finally, we also demonstrated order-of-magnitude improvements in large-scale file system re-indexing times when using our scalable monitor.

Next, we build a metadata indexing tool for large-scale HPC storage systems which has an in-tree design and uses a parallel leveled partitioning approach to partition the file system namespace into disjoint sub-trees which can be indexed in parallel. Our indexer maintains an internal metadata index database which uses a 2-level database sharding technique to increase indexing and querying performance. It also uses a changelog-based approach to keep track of the metadata changes and re-index the file system. Our design is evaluated on a 4.8 TB Lustre storage system and is compared with state-of-the-art GUFI and Robinhood engines. Our indexer improves the indexing performance by 69% and the querying performance by 91% with optimal resource utilization.

Finally, this dissertation builds an "end-to-end control plane" to optimize HPC storage systems by providing efficient load balancing across storage servers. Our proposed system provides global view of the system, enables coordination between the clients and servers, and handles the performance degradation due to resource contention by considering operations on both clients as well as servers. Our implementation provides a balanced distribution of load over OSTs and OSSs in the Lustre file system, and is able to handle both PFL and non-PFL layouts for files. We evaluated our system on a real Lustre testbed using two representative benchmarks—IOR and HACC-I/O—with multiple stripe counts of files as well as SSF and FPP accesses. Compared to the default Lustre RR policy, our design provides up to 33% improvement in balancing the load. Moreover, we also observed an I/O performance improvement of up to 43% for reads without affecting the performance for writes. Finally, the transparent design of our load balancer makes it attractive for adoption in real-world deployments.

8.2 Future Directions

This dissertation is focused on practical problems that exist in HPC storage systems. We are particularly interested in designing systems with high efficiency and flexibility and better security, and extend our understanding in cyber-physical systems and ubiquitous computing. In the following, We discuss several future directions as an extension to this dissertation.

8.2.1 ML for I/O and I/O for ML

With the rise of machine learning (ML) frameworks, there is an increasing need for optimizations in ML workloads in the HPC environment. Therefore, there is a need to analyze the I/O patterns in ML workloads and optimize HPC storage systems for ML. Additionally, with the rise in the popularity in deep learning and other ML frameworks, the optimizations should involve using ML technologies to use the enormous amounts of I/O traces to predict I/O requests from a diverse set of applications and then make HPC storage systems efficient and application-attuned.

8.2.2 Edge Computing and Federated Learning

Internet of Things holds tremendous advantages for human society. As computing capacity of Edge devices is increasing, trend to perform computing on the Edge devices is also gaining traction. With the rise in popularity of federated learning, existing edge computing frameworks lacks in optimizing I/O for such huge influx of real-time data. Also, the heterogeneity in IoT resources add additional challenge in optimization. We need to think about novel approaches to optimize I/O in a very very minimalistic resource utilization manner for the emerging IoT.

8.2.3 Containers in HPC

Containers are experiencing massive growth as the deployment unit of choice in a broad range of applications from enterprise to web services. Containers offer highly desirable features: they are lightweight, comprehensively capture dependencies into easy-to-deploy images, provide application portability, and can be scaled to meet application demands. Modern DL/ML software stacks are complex and bespoke, with no two setups exactly the same. Therefore, the use of containers is being explored in the HPC environment, and have produced benefits for large scale image processing, DL/ML workloads. In addition to meeting the computing needs of HPC applications, containers have to support the dynamic I/O requirements and scale, which introduce new challenges for data storage and access [18]. Therefore, there needs to be analysis and optimization frameworks for effective use of containers in HPC.

Bibliography

[1] ATLAS project. https://iopscience.iop.org/article/10.1088/1748-0221/3/08/S08003. Accessed: June 20 2020.

[2] Aurora Supercomputer. https://aurora.alcf.anl.gov/. Accessed: August 26 2019.

[3] BorgFS. https://www.snia.org/educational-library/borgfs-file-system-metadata-index-search-2014. Accessed: December 7 2019.

[4] Ceph User Survey, . https://ceph.io/ceph-blog/ceph-user-survey-2018-results/. Accessed: June 20 2020.

[5] Ceph Users, . https://ceph.io/users/. Accessed: June 20 2020.

[6] Cray - Clusterstor. https://www.cray.com/products/storage/clusterstor. Accessed: June 20 2020.

[7] Flame Graph. http://www.brendangregg.com/flamegraphs.html. Accessed: December 7 2019.

[8] FS_Mark. https://sourceforge.net/projects/fsmark/. Accessed: June 20 2020.

[9] GUFI. https://github.com/mar-file-system/GUFI. Accessed: November 30 2019.

[10] Ian Shields, IBM - Monitor Linux file system events with inotify. https://developer.ibm.com/tutorials/l-inotify/. Accessed: March 7 2019.

[11] LFS Find. http://manpages.ubuntu.com/manpages/precise/man1/lfs.1.html. Accessed: December 10 2019.

[12] OpenSFS and EOFS - Lustre file system. http://lustre.org/. Accessed: March 23 2019.

[13] MDTest. https://github.com/hpc/ior/blob/master/src/mdtest.c. Accessed: June 20 2020.

[14] NERSC Report. https://www.nersc.gov/news-publications/nersc-news/nersc-center-news/2017/new-storage-2020-report-outlines-future-hpc-storage-vision/. Accessed: Nov 30 2019.

[15] MezzFS — Mounting object storage in Netflix's media processing platform. https://netflixtechblog.com/mezzfs-mounting-object-storage-in-netflixs-media-processing-platform-cda01c446ba. Accessed: June 20 2020.

[16] Top 500 List. https://www.top500.org/lists/2019/11/. Accessed: November 30 2019.

[17] Why Ceph? https://searchstorage.techtarget.com/feature/5-Ceph-storage-questions-answered-and-explained. Accessed: June 20 2020.

[18] Subil Abraham, Arnab K Paul, Redwan Ibne Seraj Khan, and Ali R Butt. On the use of containers in high performance computing environments. In *2020 IEEE International Conference on Cloud Computing (CLOUD)*, pages 1–10. IEEE, 2020.

[19] Megha Agarwal, Divyansh Singhvi, Preeti Malakar, and Suren Byna. Active learning-based automatic tuning and prediction of parallel i/o performance. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 20–29. IEEE, 2019.

[20] Ravindra K Ahuja. *Network flows: theory, algorithms, and applications*. Pearson Education, 2017.

[21] Rachana Ananthakrishnan, Ben Blaiszik, Kyle Chard, Ryan Chard, Brendan McCollam, Jim Pruyne, Stephen Rosen, Steven Tuecke, and Ian Foster. Globus platform services for data publication. In *Practice and Experience on Advanced Research Computing*, page 14. ACM, 2018.

[22] Ali Anwar. *Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning*. PhD thesis, Virginia Tech, 2018.

[23] Ali Anwar, KR Krish, and Ali R Butt. On the use of microservers in supporting hadoop applications. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 66–74. IEEE, 2014.

[24] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 7–12. ACM, 2015.

[25] Ali Anwar, Anca Sailer, Andrzej Kochut, and Ali R Butt. Anatomy of cloud monitoring and metering: A case study and open problems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 6. ACM, 2015.

[26] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Cost-aware cloud metering with scalable service management infrastructure. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 285–292. IEEE, 2015.

[27] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Scalable metering for an affordable it cloud service management. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 207–212. IEEE, 2015.

[28] Ali Anwar, Salman A Baset, Andrzej P Kochut, Hui Lei, Anca Sailer, and Alla Segal. Scalable metering for cloud service management based on cost-awareness, March 31 2016. US Patent App. 14/871,443.

[29] Ali Anwar, Yue Cheng, and Ali R Butt. Towards managing variability in the cloud. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1081–1084. IEEE, 2016.

[30] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 177–188. ACM, 2016.

[31] Ali Anwar, Yue Cheng, Hai Huang, and Ali Raza Butt. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *HotStorage*, 2016.

[32] Ali Anwar, Andrzej Kochut, Anca Sailer, Charles O Schulz, and Alla Segal. Dynamic metering adjustment for service management of computing platform, March 31 2016. US Patent App. 14/926,384.

[33] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R Butt. bespo kv: application tailored scale-out key-value stores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 2. IEEE Press, 2018.

[34] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, and Ali. R. Butt. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies*, page 265, 2018.

[35] Ali Anwar, Lukas Rupprecht, Dimitris Skourtis, and Vasily Tarasov. Challenges in storing docker images. *login Usenix Mag.*, 44(3), 2019.

[36] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R Butt. Customizable scale-out key-value stores. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2081–2096, 2020.

[37] Apple. File system events. https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/FSEvents_ProgGuide/UsingtheFSEventsFramework/UsingtheFSEventsFramework.html, 2012. Accessed: Sept, 2018.

[38] Ayse Bagbaba. Improving collective i/o performance with machine learning supported auto-tuning. In *The Fifteenth International Workshop on Automatic Performance Tuning*, 2020.

[39] Babak Behzad, Surendra Byna, and Marc Snir. Optimizing i/o performance of hpc applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)*, 5 (4):1–27, 2019.

[40] Stefan Berchtold, Christian Böhm, Daniel A. Keim, and Hans-Peter Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, pages 78–86, New York, NY, USA, 1997. ACM. ISBN 0-89791-910-6. doi: 10.1145/263661.263671. URL http://doi.acm.org/10.1145/263661.263671.

[41] Thomas William Bereiter. Software auditing mechanism for a distributed computer enterprise environment, May 19 1998. US Patent 5,754,763.

[42] Tim Bisson, Yuvraj Patel, and Shankar Pasupathy. Designing a fast file system crawler with incremental differencing. *ACM SIGOPS Operating Systems Review*, 46(3):11–19, 2012.

[43] Arthur S. Bland, Jack C. Wells, Otis E. Messer, Oscar R. Hernandez, and James H. Rogers. Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory. In *Proceedings of Cray User Group Conference (CUG 2012)*, May 2012.

[44] Buddy Bland. Titan-early experience with the titan system at oak ridge national laboratory. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 2189–2211. IEEE, 2012.

[45] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control.* John Wiley & Sons, 2015.

[46] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Laboratory (LANL), 2012.

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

[48] Peter J Brockwell, Richard A Davis, and Matthew V Calder. *Introduction to time series and forecasting*, volume 2. Springer, 2002.

[49] Kirk W Cameron, Ali Anwar, Yue Cheng, Li Xu, Bo Li, Uday Ananth, Thomas Lux, Yili Hong, Layne T Watson, and Ali R Butt. Moana: Modeling and analyzing i/o variability in parallel system experimental design. 2018.

[50] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11. IEEE, 2009.

[51] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale i/o workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[52] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[53] A Fernández Casanı, D Barberis, A Favareto, C Garcıa Montoro, S González de la Hoz, J Hrivnác, F Prokoshin, J Salt, and J Sánchez. ATLAS EventIndex general dataflow and monitoring infrastructure. *Journal of Physics Conference Series*, 898(6):062010, 2017.

[54] Zheng Chai, Hannan Fayyaz, Zeshan Fayyaz, Ali Anwar, Yi Zhou, Nathalie Baracaldo, Heiko Ludwig, and Yue Cheng. Towards taming the resource and data heterogeneity in federated learning. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, pages 19–21, 2019.

[55] Zheng Chai, Ahsan Ali, Syed Zawad, Stacey Truex, Ali Anwar, Nathalie Baracaldo, Yi Zhou, Heiko Ludwig, Feng Yan, and Yue Cheng. Tifl: A tier-based federated learning system. *To appear in ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2020.

[56] Ryan Chard, Kyle Chard, Jason Alt, Dilworth Y Parkinson, Steve Tuecke, and Ian Foster. Ripple: Home automation for research data management. In *37th IEEE International Conference on Distributed Computing Systems*, 2017.

[57] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. funcx: A federated function serving fabric for science. *arXiv preprint arXiv:2005.04215*, 2020.

[58] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. ACM EuroSys*, 2015.

[59] Yue Cheng. *Workload-aware efficient storage systems.* PhD thesis, Virginia Tech, 2017.

[60] Yue Cheng, Zheng Chai, and Ali Anwar. Characterizing co-located datacenter workloads: An alibaba case study. *arXiv preprint arXiv:1808.02919*, 2018.

[61] John Cieslewicz and Kenneth A. Ross. Data partitioning on chip multiprocessors. In *Proc. ACM Data Management on New Hardware*, 2008.

[62] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. Lazybase: Trading freshness for performance in a scalable database. In *EuroSys*, pages 169–182, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168854. URL http://doi.acm.org/10.1145/2168836.2168854.

[63] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20 (3):273–297, 1995.

[64] Enrico M. Crisostomo. fswatch. https://github.com/emcrisostomo/fswatch, 2013. Accessed: Sept, 2018.

[65] Breno Dantas Cruz, Arnab K Paul, and Eli Tilevich. Stargazer: A deep learning approach for estimating the performance of edge-based clustering applications. In *2020 IEEE International Conference on Smart Data Services (SMDS)*, pages 1–10. IEEE, 2020.

[66] Arjun Datta and Arnab Kumar Paul. Online compiler as a cloud service. In *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, pages 1783–1786. IEEE, 2014.

[67] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI*, 2004.

[68] Shyam C Deshmukh and Sudarshan S Deshmukh. Improved load balancing for distributed file system using self acting and adaptive loading data migration process. In *4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions), 2015*, pages 1–6. IEEE, 2015.

[69] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, Nawab Ali, and P Sadayappan. Integrating parallel file systems with object-based storage devices. In *SC'07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10. IEEE, 2007.

[70] Tim d'Hondt, Anna Wilbik, Paul Grefen, Heiko Ludwig, Natalie Baracaldo, and Ali Anwar. Using bpm technology to deploy and manage distributed analytics in collaborative iot-driven business scenarios. In *Proceedings of the 9th International Conference on the Internet of Things*, pages 1–8, 2019.

[71] Bin Dong, Xiuqiao Li, Qimeng Wu, Limin Xiao, and Li Ruan. A dynamic and adaptive load balancing strategy for parallel file system with large-scale i/o servers. *Journal of Parallel and Distributed Computing*, 72(10):1254–1268, 2012.

[72] Jack Dongarra, Hans Meuer, and Erich Strohmaier. Top500 supercomputing sites. http://www.top500.org, 2016.

[73] Facebook. Watchman: A file watching service. https://facebook.github.io/watchman/, 2015. Accessed: Sept, 2018.

[74] Argonne Leadership Computing Facility. Intrepid system. URL https://www.alcf.anl.gov/intrepid. Accessed: May 12 2019.

[75] Ian Foster, Ben Blaiszik, Kyle Chard, and Ryan Chard. Software Defined Cyberinfrastructure. In *The 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[76] Rohan Gandhi, Di Xie, and Y. Charlie Hu. Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *Proc. USENIX ATC*, 2013.

[77] Dominic Giampaolo. *Practical file system design with the Be file system*. Morgan Kaufmann Publishers Inc., 1998.

[78] Alexandra Glagoleva and Archana Sathaye. Load balancing distributed file system servers: a rule-based approach. *Web-Enabled Systems Integration: Practices and Challenges: Practices and Challenges*, page 274, 2002.

[79] Gluster. Gluster-cloud storage for the modern data center, 2017. http://moo.nac.uci.edu/ hjm/fs/AnIntroductionToGlusterArchitectureV7110708.pdf.

[80] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. USENIX OSDI*, 2014.

[81] Raghul Gunasekaran, Sarp Oral, Jason Hill, Ross Miller, Feiyi Wang, and Dustin Leverman. Comparative i/o workload characterization of two leadership class storage clusters. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 31–36. ACM, 2015.

[82] Marios Hadjieleftheriou, Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J Tsotras. R-trees: A dynamic index structure for spatial searching. *Encyclopedia of GIS*, pages 1805–1817, 2017.

[83] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28 (1):100–108, 1979.

[84] Andrew C Harvey. *Forecasting, structural time series models and the Kalman filter*. Cambridge university press, 1990.

[85] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[86] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly, 2013.

[87] David W Hosmer Jr and Stanley Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.

[88] Windsor W Hsu and Alan Jay Smith. Characteristics of i/o traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2):347–372, 2003.

[89] Windsor W Hsu, Alan Jay Smith, and Honesty C Young. I/o reference behavior of production database workloads and the tpc benchmarks—an analysis at the logical level. *ACM Transactions on Database Systems (TODS)*, 26(1):96–143, 2001.

[90] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian. Smartstore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *SC*, pages 1–12, Nov 2009. doi: 10.1145/1654059.1654070.

[91] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. In *Proc. USENIX ATC*, 2015.

[92] InfluxData. Influxdb, . URL `https://github.com/influxdata/influxdb`. Accessed: April 1 2019.

[93] InfluxData. Telegraf, . URL `https://github.com/influxdata/telegraf`. Accessed: April 1 2019.

[94] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. ACM Eurosys*, 2007.

[95] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[96] Sunggon Kim, Alex Sim, Kesheng Wu, Suren Byna, Yongseok Son, and Hyeonsang Eom. Towards hpc i/o performance prediction through large-scale log analysis. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 77–88, 2020.

[97] Youngjae Kim, Raghul Gunasekaran, Galen M Shipman, David A Dillow, Zhe Zhang, and Bradley W Settlemyer. Workload characterization of a leadership class storage cluster. In *2010 5th Petascale Data Storage Workshop (PDSW'10)*, pages 1–5. IEEE, 2010.

[98] Donghun Koo, Jik-Soo Kim, Soonwook Hwang, Hyeonsang Eom, and Jaehwan Lee. Utilizing progressive file layout leveraging ssds in hpc cloud environments. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 90–95. IEEE, 2016.

[99] K. R. Krish, B. Wadhwa, M. S. Iqbal, M. M. Rafique, and A. R. Butt. On efficient hierarchical storage for big data processing. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 403–408, 2016.

[100] KR Krish, Ali Anwar, and Ali R Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2014.

[101] KR Krish, Ali Anwar, and Ali R Butt. [phi] sched: A heterogeneity-aware hadoop workflow scheduler. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 255–264. IEEE, 2014.

[102] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In *Proc. ACM ASPLOS*, 2008.

[103] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proc. ACM SIGMOD*, 2012.

[104] Argonne National Laboratory. Darshan - hpc i/o characterization tool. URL `https://www.mcs.anl.gov/research/projects/darshan/`. Accessed: May 12 2019.

[105] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/o performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12. IEEE, 2009.

[106] Margaret Lawson and Jay Lofstead. Using a robust metadata management system to accelerate scientific discovery at extreme scales. In *2018 IEEE/ACM PDSW-DISCS*, pages 13–23. IEEE, 2018.

[107] Thomas Leibovici. Taking back control of HPC file systems with Robinhood Policy Engine. *arXiv preprint arXiv:1505.01448*, 2015.

[108] Jonathan Lemon. Kqueue – A generic and scalable event notification facility. In *USENIX Annual Technical Conference, FREENIX Track*, pages 141–153, 2001.

[109] Andrew Leung, I Adams, and Ethan L Miller. Magellan: A searchable metadata architecture for large-scale file systems. *University of California, Santa Cruz, Tech. Rep. UCSC-SSRC-09-07*, 2009.

[110] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference*, volume 1, pages 5–2, 2008.

[111] Andrew W Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *FAST*, volume 9, pages 153–166, 2009.

[112] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proc. ACM International Conference on Computing Frontiers*, 2015.

[113] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for mapreduce workflows. In *Proc. VLDB*, 2012.

[114] Seung-Hwan Lim, Hyogi Sim, Raghul Gunasekaran, and Sudharshan S Vazhkudai. Scientific user behavior and data-sharing trends in a petascale file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 46. ACM, 2017.

[115] Linux. inotify-tools. https://github.com/rvoicilas/inotify-tools/, 2010. Accessed: Sept, 2018.

[116] Linux. fanotify. http://man7.org/linux/man-pages/man7/fanotify.7.html, 2017. Accessed: Sept, 2018.

[117] Michael Littley, Ali Anwar, Hannan Fayyaz, Zeshan Fayyaz, Vasily Tarasov, Lukas Rupprecht, Dimitrios Skourtis, Mohamed Mohamed, Heiko Ludwig, Yue Cheng, and Ali R Butt. Bolt: Towards a scalable docker registry via hyperconvergence. In *IEEE International Conference on Cloud Computing*, 2019.

[118] Jinjun Liu, Dan Feng, Yu Hua, Bin Peng, and Zhenhua Nie. Using provenance to efficiently improve metadata searching performance in storage systems. *Future Generation Computer Systems*, 50:99–110, 2015.

[119] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Server-side log data analytics for i/o workload characterization and coordination on large shared storage systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 819–829. IEEE, 2016.

[120] LLNL. Hacc i/o benchmark summary, 2017. URL https://asc.llnl.gov/CORAL-benchmarks/Summaries/HACC_IO_Summary_v1.0.pdf.

[121] LLNL. Ior benchmark, 2017. https://asc.llnl.gov/sequoia/ benchmarks/IOR_summary_v1.0.pdf.

[122] Glenn K Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J Wright. A year in the life of a parallel file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 74. IEEE Press, 2018.

[123] Glenn K Lockwood, Nicholas J Wright, Shane Snyder, Philip Carns, George Brown, and Kevin Harms. Tokio on clusterstor: connecting standard tools to enable holistic i/o performance analysis. In *2018 Cray User Group*, 2018.

[124] Robert Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005(139):8, 2005.

[125] Lustre. Lustre - distributed name space, . URL `http://wiki.lustre.org/Lustre_Metadata_Service_(MDS)`. Accessed: July 15 2020.

[126] Lustre. Lustre jobstats, . URL `http://doc.lustre.org/lustre_manual.xhtml#dbdoclet.jobstats`. Accessed: April 1 2019.

[127] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. A Multi-Level Approach for Understanding I/O Activity in HPC Applications. In *Proc. CLUSTER*. IEEE, September 2013. doi: 10.1109/CLUSTER.2013.6702690.

[128] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A multiplatform study of i/o behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 33–44. ACM, 2015.

[129] J. F. Martinez and E. Ipek. Dynamic multicore resource management: A machine learning approach. *IEEE Micro*, 29(5):8–17, 2009.

[130] Ryan McKenna, Stephen Herbein, Adam Moody, Todd Gamblin, and Michela Taufer. Machine learning predictions of runtime and io traffic on high-end clusters. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 255–258. IEEE, 2016.

[131] Microsoft. FileSystemWatcher. `https://docs.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher?redirectedfrom=MSDN&view=netframework-4.7.2`, 2010. Accessed: Sept, 2018.

[132] Rich Miller. Google using machine learning to boost data center efficiency — data center knowledge, 2014. URL `http://www.datacenterknowledge.com/archives/2014/05/28/google-using-machine-learning-boost-data-center-efficiency/2/`.

[133] Ross Miller, Jason Hill, David A Dillow, Raghul Gunasekaran, Galen M Shipman, and Don Maxwell. Monitoring tools for large scale systems. In *Cray User Group Conference*, 2010.

[134] Rick Mohr, Michael Brim, Sarp Oral, and Andreas Dilger. Evaluating progressive file layouts for lustre. In *Cray User Group Conference (CUG 2016)*, 2016.

[135] Esteban Molina-Estolano, Carlos Maltzahn, and Scott Brandt. Dynamic load balancing in ceph. 2008.

[136] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *Proc. ACM SIGMOD*, 2011.

[137] Sarah Neuwirth. *Accelerating Network Communication and I/O in Scientific High Performance Computing Environments.* PhD thesis, Heidelberg University, Germany, December 2018.

[138] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Bruening. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance. In *Proc. ICPADS*. IEEE, 2017.

[139] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, C Sclatter Ellis, and Michael L Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, 1996.

[140] NS-3. Network simulator, 2017. http://code.nsnam.org/.

[141] Michael A Olson et al. The design and implementation of the inversion file system. In *USENIX Winter*, pages 205–218, 1993.

[142] OpenSFS and EOFS. Lustre file system. http://lustre.org/. Accessed: November 2 2019.

[143] Sarp Oral, Feiyi Wang, David Dillow, Galen M Shipman, Ross Miller, and Oleg Drokin. Efficient object storage journaling in a distributed parallel file system. In *FAST*, volume 10, pages 1–12, 2010.

[144] Sarp Oral, James Simmons, Jason Hill, Dustin Leverman, Feiyi Wang, Matt Ezell, Ross Miller, Douglas Fuller, Raghul Gunasekaran, Youngjae Kim, et al. Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 217–228. IEEE, 2014.

[145] Sarp Oral, Sudharshan S Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross Miller, Dustin Leverman, Scott Atchley, et al. End-to-end i/o portfolio for the summit supercomputing ecosystem. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[146] Aleatha Parker-Wood, Christina Strong, Ethan L Miller, and Darrell DE Long. Security aware partitioning for efficient file system search. In *2010 IEEE 26th MSST*, pages 1–14. IEEE, 2010.

[147] Barbara K Pasquale and George C Polyzos. A static analysis of i/o characteristics of scientific applications in a production workload. In *Supercomputing'93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 388–397. IEEE, 1993.

[148] Tirthak Patel, Suren Byna, Glenn K Lockwood, Nicholas J Wright, Philip Carns, Robert Ross, and Devesh Tiwari. Uncovering access, reuse, and sharing characteristics of i/o-intensive files on large-scale production {HPC} systems. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 91–101, 2020.

[149] Swapnil Patil and Garth A Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *FAST*, pages 13–13, 2011.

[150] Arnab K Paul, Arpit Goyal, Feiyi Wang, Sarp Oral, Ali R Butt, Michael J Brim, and Sangeetha B Srinivasa. I/O load balancing for big data HPC applications. In *International Conference on Big Data*, pages 233–242. IEEE, 2017.

[151] Arnab K Paul, Arpit Goyal, Feiyi Wang, Sarp Oral, Ali R Butt, Michael J Brim, and Sangeetha B Srinivasa. I/o load balancing for big data hpc applications. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 233–242. IEEE, 2017.

[152] Arnab K Paul, Steven Tuecke, Ryan Chard, Ali R Butt, Kyle Chard, and Ian Foster. Toward scalable monitoring on large-scale storage for software defined cyberinfrastructure. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 49–54, 2017.

[153] Arnab K Paul, Ryan Chard, Kyle Chard, Steven Tuecke, Ali R Butt, and Ian Foster. Fsmonitor: Scalable file system monitoring for arbitrary storage systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[154] Arnab K Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R Butt. Understanding hpc application i/o behavior using system level statistics, 2019.

[155] Arnab K Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R Butt. Improving i/o performance of hpc applications using intra-job scheduling. In *2019 Work-In-Progress Proceedings of the Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 1 – 1, 2019.

[156] Arnab K Paul, Ryan Chard, Kyle Chard, Ali R Butt, and Ian Foster. Storm: File system monitoring for large scale storage systems. In *In Submission*, pages 1–12. IEEE, 2020.

[157] Arnab K Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R Butt. Understanding hpc application i/o behavior using system level statistics. In *In Submission*, pages 1–10, 2020.

[158] Arnab K Paul, Bharti Wadhwa, Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ali R Butt. Resource contention aware load balancing for large-scale parallel file systems. In *In Submission*, pages 1–12, 2020.

[159] Arnab K Paul, Brian Wang, Nathan Rutman, Cory Spitz, and Ali R Butt. Efficient metadata indexing for hpc storage systems. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 162–171. IEEE, 2020.

[160] Arnab Kumar Paul. *Dynamic virtual machine placement in cloud computing*. PhD thesis, 2015.

[161] Arnab Kumar Paul and Bibhudatta Sahoo. Dynamic virtual machine placement in cloud computing. In *Resource Management and Efficiency in Cloud Computing Environments*, pages 136–167. IGI Global, 2017.

[162] Arnab Kumar Paul, Sourav Kanti Addya, Bibhudatta Sahoo, and Ashok Kumar Turuk. Application of greedy algorithms to virtual machine distribution across data centers. In *2014 Annual IEEE India Conference (INDICON)*, pages 1–6. IEEE, 2014.

[163] Arnab Kumar Paul, Wenjie Zhuang, Luna Xu, Min Li, M Mustafa Rafique, and Ali R Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 110–119. IEEE, 2016.

[164] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. ACM SIGMOD*, 2012.

[165] Python. Watchdog. https://pypi.org/project/watchdog/, 2010. Accessed: Sept, 2018.

[166] Yingjin Qian, Eric Barton, Tom Wang, Nirant Puntambekar, and Andreas Dilger. A novel network request scheduler for a large scale storage system. *Computer Science - Research and Development*, 23(3):143–148, 2009. ISSN 1865-2042. doi: 10.1007/s00450-009-0073-9. URL http://dx.doi.org/10.1007/s00450-009-0073-9.

[167] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proc. ACM International Conference on Extending Database Technology*, 2013.

[168] Dino Quintero, Luis Bolinches, Puneet Chaudhary, Willard Davis, Steve Duersch, Carlos Henrique Fachim, Andrei Socoliuc, Olaf Weiser, et al. *IBM Spectrum Scale (formerly GPFS)*. IBM Redbooks, 2017.

[169] Subhash Saini, Jason Rappleye, Johnny Chang, David Barker, Piyush Mehrotra, and Rupak Biswas. I/o performance characterization of lustre and nasa applications on pleiades. In *2012 19th International Conference on High Performance Computing*, pages 1–10. IEEE, 2012.

[170] Ramesh R Sarukkai. Link prediction and path analysis using markov chains. *Computer Networks*, 33(1):377–386, 2000.

[171] Andrea Schaerf, Yoav Shoham, and Moshe Tennenholtz. Adaptive load balancing: A study in multi-agent learning. *Journal of Artificial Intelligence Research*, 2:475–500, 1995.

[172] SchedMD. Slurm workload manager. URL https://slurm.schedmd.com/overview.html. Accessed: April 1 2019.

[173] C. Selvakumar, G. J. Rathanam, and M. R. Sumalatha. Pdds - improving cloud data storage security using data partitioning technique. In *Proc. IEEE International Advance Computing Conference*, 2013.

[174] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proc. ACM SIGMOD*, 2013.

[175] Galen Shipman, David Dillow, Sarp Oral, Feiyi Wang, Douglas Fuller, Jason Hill, and Zhe Zhang. Lessons learned in deploying the world's largest scale lustre file system. In *The 52nd Cray user group conference*, 2010.

[176] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proc. IEEE MSST*, 2010.

[177] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R Butt, and Lavanya Ramakrishnan. Analyzethis: an analysis workflow-aware storage system. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[178] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Geoffroy R Vallée, Seung-Hwan Lim, and Ali R Butt. Tagit: an integrated indexing and search service for file systems. In *SC*, page 5. ACM, 2017.

[179] Hyogi Sim, Arnab K Paul, Eli Tilevich, Ali R Butt, and Muhammad Shahzad. Cslim: automated extraction of iot functionalities from legacy c codebases. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, pages 421–426, 2019.

[180] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of ACM/IEEE SC*, 2008.

[181] Craig A.N. Soules, Kimberly Keeton, and Charles B. Morrey, III. Scan-lite: Enterprise-wide analysis on the cheap. In *EuroSys*, New York, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: 10.1145/1519065.1519079. URL http://doi.acm.org/10.1145/1519065.1519079.

[182] Toyotaro Suzumura, Yi Zhou, Natahalie Barcardo, Guangnan Ye, Keith Houck, Ryo Kawahara, Ali Anwar, Lucia Larise Stavarache, Daniel Klyashtorny, Heiko Ludwig, et al. Towards federated graph learning for collaborative financial crimes detection. *arXiv preprint arXiv:1909.12946*, 2019.

[183] Narate Taerat, Nichamon Naksinehaboon, Clayton Chandler, James Elliott, Chokchai Leangsuksun, George Ostrouchov, Stephen L Scott, and Christian Engelmann. Blue gene/l log analysis and time to interrupt estimation. In *2009 International Conference on Availability, Reliability and Security*, pages 173–180. IEEE, 2009.

[184] Douglas A Talbert and Doug Fisher. An empirical analysis of techniques for constructing and searching k-dimensional trees. In *Proceedings of the sixth ACM SIGKDD*, pages 26–33. ACM, 2000.

[185] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. Someta: Scalable object-centric metadata management for high performance computing. In *CLUSTER*, pages 359–369. IEEE, 2017.

[186] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 1–11, 2019.

[187] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proc. ACM International Conference on Web Search and Data Mining*, 2014.

[188] Alan Tucker. A note on convergence of the ford-fulkerson flow algorithm. *Mathematics of Operations Research*, 2(2):143–144, 1977.

[189] A. Turcu, R. Palmieri, B. Ravindran, and S. Hirve. Automated data partitioning for highly scalable and strongly consistent transactions. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):106–118, 2016. ISSN 1045-9219.

[190] UMass. Umass trace repository, 2017. http://traces.cs.umass.edu/ index.php/Storage/Storage.

[191] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. Logbase: A scalable log-structured database system in the cloud. In *Proc. ACM VLDB*, 2012.

[192] B. Wadhwa and A. Verma. Carbon efficient vm placement and migration technique for green federated cloud datacenters. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2297–2302, 2014.

[193] B. Wadhwa and A. Verma. Energy saving approaches for green cloud computing: A review. In *2014 Recent Advances in Engineering and Computational Sciences (RAECS)*, pages 1–6, 2014.

[194] B. Wadhwa and A. Verma. Energy and carbon efficient vm placement and migration technique for green cloud datacenters. In *2014 Seventh International Conference on Contemporary Computing (IC3)*, pages 189–193, 2014.

[195] B. Wadhwa, S. Byna, and A. R. Butt. Toward transparent data management in multi-layer storage hierarchy of hpc systems. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 211–217, 2018.

[196] Bharti Wadhwa, Arnab K Paul, Sarah Neuwirth, Feiyi Wang, Sarp Oral, Ali R Butt, Jon Bernard, and Kirk W Cameron. iez: Resource contention aware load balancing for large-scale parallel file systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 610–620. IEEE, 2019.

[197] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, 2020.

[198] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S Vazhkudai. Improving large-scale storage system performance via topology-aware and balanced data placement. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 656–663. IEEE, 2014.

[199] Feng Wang, Qin Xin, Bo Hong, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Tyce T McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, 2004.

[200] Teng Wang, Shane Snyder, Glenn Lockwood, Philip Carns, Nicholas Wright, and Suren Byna. Iominer: Large-scale analytics framework for gaining knowledge from i/o logs. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 466–476. IEEE, 2018.

[201] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[202] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of ACM/IEEE SC*, 2006.

[203] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proc. ACM ISCA*, 2013.

[204] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proc. ACM SoCC*, 2011.

[205] MR Wyatt, S Herbein, T Gamblin, A Moody, A Dong, and M Taufer. From job scripts to resource predictions: Paving the path to next-generation hpc schedulers. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[206] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proc. ACM SIGMOD*, 2013.

[207] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 13–23, 2019.

[208] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, et al. End-to-end i/o monitoring on a leading supercomputer. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 379–394, 2019.

[209] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Perform. Eval. Rev.*, 40(4):23–32, 2013. ISSN 0163-5999.

[210] Xin Yang, Qi Liu, BC Yin, Qiang Zhang, DS Zhou, and XP Wei. k-d tree construction designed for motion blur. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations*, pages 113–119. Eurographics Association, 2017.

[211] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proc. USENIX HotCloud*, 2010. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

[212] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX NSDI*, 2012.

[213] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 61–70. IEEE, 2014.

[214] Dongfang Zhao, Kan Qiao, Zhou Zhou, Tonglin Li, Zhihan Lu, and Xiaohua Xu. Toward efficient and flexible metadata indexing of big data systems. *IEEE Transactions on Big Data*, 3(1):107–117, 2017.

[215] N. Zhao, J. Wan, J. Wang, and C. Xie. Greencht: A power-proportional replication scheme for consistent hashing based key value storage systems. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, 2015.

[216] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, 2019.

[217] N. Zhao, V. Tarasov, A. Anwar, L. Rupprecht, D. Skourtis, A. Warke, M. Mohamed, and A. Butt. Slimmer: Weight loss secrets for docker registries. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 517–519, 2019.

[218] Nan-nan Zhao, Ji-guang Wan, Jun Wang, and Chang-sheng Xie. A reliable power management scheme for consistent hashing based distributed key value storage systems. *Frontiers of Information Technology and Electronic Engineering*, 17:994–1007, 10 2016. doi: 10.1631/FITEE.1601162.

[219] Nannan Zhao, Ali Anwar, Yue Cheng, Mohammed Salman, Daping Li, Jiguang Wan, Changsheng Xie, Xubin He, Feiyi Wang, and Ali Raza Butt. Chameleon: An adaptive wear balancer for flash clusters. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1163–1172, 2018.

[220] Nannan Zhao, Ali Anware, Yue Cheng, Mohammed Salman, Daping Li, Jiguang Wan, Changsheng Xie, Xubin He, Feiyi Wang, and Ali Butt. Chameleon: An adaptive wear balancer for flash clusters. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1163–1172. IEEE, 2018.

[221] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2019.

[222] Nannan Zhao, Vasily Tarasov, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit Warke, Mohamed Mohamed, and Ali Butt. Slimmer: Weight loss secrets for docker registries. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 517–519. IEEE, 2019.

[223] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R. Butt. Duphunter: Flexible high-performance deduplication for docker registries. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 769–783. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/zhao.

[224] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R Butt. Duphunter: Flexible high-performance deduplication for docker registries. *To appear in USENIX Annual Technical Conference (ATC 20)*, 2020.

[225] Tiezhu Zhao, Verdi March, Shoubin Dong, and Simon See. Evaluation of a performance model of lustre file system. In *2010 Fifth Annual ChinaGrid Conference*, pages 191–196. IEEE, 2010.

[226] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on multicore and multi-gpu platforms using functional performance models. *IEEE Transactions on Computers*, 64(9):2506–2518, 2015.

[227] Peipei Zhou, Zhenyuan Ruan, Zhenman Fang, Megan Shand, David Roazen, and Jason Cong. Doppio: I/o-aware performance analysis, modeling and optimization for in-memory computing framework. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 22–32. IEEE, 2018.

[228] Sergi Àlvarez. Fsmon. https://github.com/nowsecure/fsmon, 2016. Accessed: Sept, 2018.