Florida International University
FIU Digital Commons

FIU Electronic Theses and Dissertations | University Graduate School

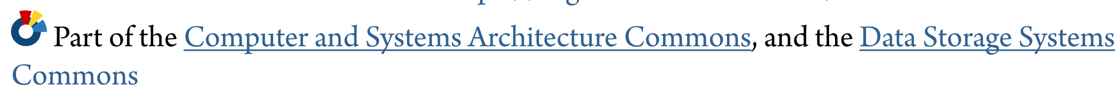
3-18-2016

Storage Management of Data-intensive Computing Systems

Yiqi Xu
Florida International University, yxu006@fiu.edu

DOI: 10.25148/etd.FIDC000251

Follow this and additional works at: https://digitalcommons.fiu.edu/etd


Part of the Computer and Systems Architecture Commons, and the Data Storage Systems Commons

Recommended Citation
Xu, Yiqi, "Storage Management of Data-intensive Computing Systems" (2016). *FIU Electronic Theses and Dissertations*. 2474.
https://digitalcommons.fiu.edu/etd/2474

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

STORAGE MANAGEMENT OF DATA-INTENSIVE COMPUTING SYSTEMS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Yiqi Xu

2016

To: Interim Dean Ranu Jung
College of Engineering and Computing

This dissertation, written by Yiqi Xu, and entitled Storage Management of Data-intensive Computing Systems, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Jason Liu

Deng Pan

Gang Quan

Raju Rangaswami

Seetharami R. Seelam

Ming Zhao, Major Professor

Date of Defense: March 18, 2016

The dissertation of Yiqi Xu is approved.

Interim Dean Ranu Jung
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2016

© Copyright 2016 by Yiqi Xu

All rights reserved.

DEDICATION

To my family.

ACKNOWLEDGMENTS

I would like to express my most sincere appreciation to my advisor, Prof. Ming Zhao, for his attentive, patient, and encouraging mentoring throughout my Ph.D. study. Prof. Ming Zhao is the first faculty to grant me the research opportunities during my graduate study, with great enthusiasms and promising potentials in his ideas. Not only is he a capable individual advisor for each of his students, but he is also an effective leader cultivating a positive atmosphere in the VISA lab, where I comfortably conduct my research. He provided me with opportunities in both academia and industry to strengthen my technical skills and widen my vision, to which I am truly obliged.

I am very grateful to the other members of my dissertation committee. Prof. Raju Rangaswami instilled the knowledge of basic and modern storage systems in his class. Prof. Jason Liu, Gang Quan, and Deng Pan gave me precious feedbacks to my dissertation topic and presentation for deeper understanding and improvements. Dr. Seetharami Seelam has been a great supporter and collaborator, and till today I am still benefitting from the insights that he has brought to the research.

I am also indebted to Prof. Binyu Zang, who opened a world of possibilities to me in Fudan University and supported my study overseas. I would like to show my gratitude to Dr. Brent Welch, who carefully mentored and guided me during my first internship in Panasas Inc. He set me examples in both technical expertise and personal character. I appreciate the internship opportunities and the graduate fellowship granted by VMware Inc to support my research and study. VMware also provided me with two fruitful, unforgettable summers of indulging in the state-of-the-art technologies and working with researchers alike.

My genuine thanks also go to my fellow VISA lab members. I had many birthday surprise parties in the VISA lab which I can still vividly recall. The support we gave each other, the hard working time we endured together, and the dreams we share have bonded

us like no other. I am especially thankful to Dulcardo Arteaga, who introduced me to the piece of software that I still use for my dissertation today. I thank the earnest friendship from Douglas Otstott, without whom the lab is really different.

Finally, and most importantly, I owe my family everything for what I can achieve today and in the future. My parents' and grandparents' loving care made me believe in excellence, love, and family, which supported me through difficult times. My twin brother Yifei and I encourage each other even when we are worlds apart, whenever either one of us is in a hard time. My fiancé Liang Sun has been a godsend to me since the beginning of my Ph.D. pursuit. I am lucky to share every bit of happiness in life with him. His total commitment made me a better person and I look forward to starting a new family with him. They are my motivation and inspiration. I dedicate this dissertation to them.

ABSTRACT OF THE DISSERTATION

STORAGE MANAGEMENT OF DATA-INTENSIVE COMPUTING SYSTEMS

by

Yiqi Xu

Florida International University, 2016

Miami, Florida

Professor Ming Zhao, Major Professor

Computing systems are becoming increasingly data-intensive because of the explosion of data and the needs for processing the data, and storage management is critical to application performance in such data-intensive computing systems. However, existing resource management frameworks in these systems lack the support for storage management, which causes unpredictable performance degradations when applications are under I/O contention. Storage management of data-intensive systems is a challenging problem because I/O resources cannot be easily partitioned and distributed storage systems require scalable management. This dissertation presents the solutions to address these challenges for typical data-intensive systems including high-performance computing (HPC) systems and big-data systems.

For HPC systems, the dissertation presents vPFS, a performance virtualization layer for parallel file system (PFS) based storage systems. It employs user-level PFS proxies to interpose and schedule parallel I/Os on a per-application basis. Based on this framework, it enables SFQ(D)+, a new proportional-share scheduling algorithm which allows diverse applications with good performance isolation and resource utilization. To manage an HPC system's total I/O service, it also provides two complementary synchronization schemes to coordinate the scheduling of large numbers of storage nodes in a scalable manner.

For big-data systems, the dissertation presents IBIS, an interposition-based big-data I/O scheduler. By interposing the different I/O phases of big-data applications, it schedules the I/Os transparently to the applications. It enables a new proportional-share scheduling algorithm, SFQ(D2), to address the dynamics of the underlying storage by adaptively adjusting the I/O concurrency. Moreover, it employs a scalable broker to coordinate the distributed I/O schedulers and provide proportional sharing of a big-data system's total I/O service.

Experimental evaluations show that these solutions have low-overhead and provide strong I/O performance isolation. For example, vPFS' overhead is less than 3% in throughput and it delivers proportional sharing within 96% of the target for diverse workloads; and IBIS provides up to 99% better performance isolation for WordCount and 30% better proportional slowdown for TeraSort and TeraGen than native YARN.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION 1
1.1 HPC Storage Systems 4
1.2 Big-data Storage Systems 8
1.3 Outline 11

2. RELATED WORK 12
2.1 HPC Storage Systems 12
2.2 Big-data Storage Systems 13
2.3 Related Storage Management Work 15
2.3.1 HPC Storage Management 15
2.3.2 Big-data Storage Management 17
2.3.3 Management of Other Types of Storage Systems 19

3. HPC STORAGE SYSTEMS MANAGEMENT 21
3.1 Introduction 21
3.2 Motivation 23
3.3 Parallel Storage System Virtualization 27
3.4 Fair Share of Parallel Storage I/Os 29
3.4.1 Review of SFQ and SFQ(D) 30
3.4.2 Variable Cost I/O Depth Allocation 32
3.4.3 Backfill I/O Dispatching 37
3.5 Parallel Storage Distributed Scheduling Coordination 39
3.5.1 Total-Service Proportional Sharing 39
3.5.2 Threshold-driven Global Scheduling Synchronization 41
3.5.3 Layout-driven Total-Service Proportional Sharing 43
3.6 Parallel Storage Metadata Scheduling 45
3.7 Experimental Evaluation 49
3.7.1 Setup 49
3.7.2 IOR with Various Access Patterns 52
3.7.3 IOR with Dynamic Arrivals 55
3.7.4 BTIO vs. IOR 57
3.7.5 WRF vs. IOR 60
3.7.6 Metadata Scheduling 62
3.7.7 Overhead 65
3.7.8 Cost of Implementation 72
3.8 Summary 73

4. BIG-DATA STORAGE SYSTEMS MANAGEMENT 75
4.1 Introduction 75
4.2 Motivation 77
4.3 Interposed I/O Scheduling 80
4.4 Proportional I/O Sharing 83
4.5 Distributed I/O Scheduling Coordination 86
4.6 Multi-framework I/O Scheduling 89
4.7 Experimental Evaluation 91
4.7.1 Setup 91
4.7.2 Performance Isolation (WordCount) 92
4.7.3 Performance Isolation (Facebook2009) 97
4.7.4 Multi-framework I/O Scheduling 99
4.7.5 Proportional Slowdown 103
4.7.6 Coordinated Scheduling 104
4.7.7 Overhead 106
4.8 Summary 107

5. CONCLUSIONS AND FUTURE WORK 109
5.1 Conclusions 109
5.2 Future Work 113
5.2.1 Latency-driven I/O Scheduling 113
5.2.2 Coordinated Storage and Network I/O Management 115
5.2.3 End-to-end Application Performance Guarantee 116

BIBLIOGRAPHY 118

VITA 129

LIST OF FIGURES

FIGURE PAGE

2.1 The architecture of a parallel storage system 13
2.2 The architecture of a big-data storage system 14
3.1 BTIO performance slowdown 25
3.2 The architecture of PFS virtualization 28
3.3 Throughput models for a PFS storage node 34
3.4 Latency models for a PFS storage node 35
3.5 Metadata I/Os and data I/Os comparison 47
3.6 IOR write vs. IOR write 53
3.7 IOR write vs. IOR read 53
3.8 IOR write vs. IOR random read/write 54
3.9 The throughput of 8 competing IORs 55
3.10 The fairness of 8 competing IORs 56
3.11 BTIO performance restoration 58
3.12 BTIO runtime restoration 58
3.13 WRF runtime restoration 61
3.14 Combined throughput of WRF and IOR 63
3.15 Metadata I/O throughput 63
3.16 The overhead of vPFS for a data-intensive application 66
3.17 The overhead of vPFS for a metadata-intensive application 67
3.18 CPU and memory overhead of vPFS 67
3.19 I/O throughput of different synchronization schemes 70
3.20 Network traffic of different synchronization schemes 71
3.21 Application throughput of different synchronization schemes 72
4.1 I/O demands of two classic MapReduce applications 78

4.2 WordCount performance slowdown 79
4.3 Architecture of MapReduce and IBIS 81
4.4 Architecture for distributed I/O scheduling coordination 88
4.5 Integration of IBIS with YARN 90
4.6 WordCount performance slowdown with HDDs 93
4.7 Adaptation of *D* by SFQ(D2) 95
4.8 WordCount performance slowdown with SDDs 96
4.9 Cumulative distribution of Facebook2009 job runtimes 98
4.10 Performance restoration of TPC-H on Hive 101
4.11 Performance slowdown of TeraSort and TeraGen with or without IBIS 104
4.12 Performance slowdown of TeraSort and TeraGen with or without sync 105
4.13 Runtime overhead of IBIS 106

CHAPTER 1

INTRODUCTION

Data-intensive computing systems have penetrated every aspect of people's lives. Behind it is the scientific and commercial processing of massive data impacting the decision makings in companies, academics, governments and personal lives. Two types of data-intensive computing systems continue to co-exist in the modern computing environment. High Performance Computing (HPC) systems, consisting of tightly coupled compute nodes and storage nodes, are used to execute task parallelism for scientific purposes like weather forecasting, physics simulation, and the likes. Message Passing Interface (MPI) is an example of a computing framework on HPC systems. Big-data systems, comprised of more loosely coupled nodes, are used to execute data parallelism for tasks such as sorting, data mining, machine learning, etc. MapReduce is an example of a computing framework on big-data systems. The clearer definitions and the differences of the two types of systems will be explained below in details.

Both HPC systems and big-data systems are deployed for multiple users and applications to share the computing resources so that 1) the resource utilization is high, driving down the usage cost per application/user, and the users get better responsiveness of application execution; 2) the data set is reused without extra overhead to move around performing redundant I/Os, and users can also save space. As the computing needs continue to grow in data-intensive computing systems, the shared usage model results in a highly resource-competing environment. For example, Amazon provides HPC and big-data as cloud services. Hadoop version 2, YARN, provides a scheduler to encompass both MapReduce and MPI jobs.

As the number of concurrent data-intensive applications and the amount of data increase, application I/Os start to saturate the storage and interfere with each other, and storage systems become the bottleneck to application performance. Both HPC and big-

data systems' I/O amplification adds to the I/O contention in the storage systems. To counter failures in these distributed systems, HPC systems employ defensive I/Os such as checkpointing to restart an application from where it fails, and big-data systems replicate persistent data by a factor of k, which grows with the scale of the storage system. Both mechanisms aggravate the I/O contention on the storage. The storage systems can be scaled-out, but the compute to storage node ratio is still high, rendering the storage subsystem a highly contended component.

Therefore, the lack of I/O performance isolation in the data-intensive computing systems causes severe storage interference which compromises the performance target set by other resource managers proposed or implemented in a large body of works. Failure to provide applications with guaranteed performance has consequences. Data-intensive applications must complete in bounded time so as to get meaningful results. For example, weather forecast data is much less useful when the forecasted time has passed. Paid user in a big-data system also require a predictable runtime even though the job is not time-sensitive, and the provider may get penalized in revenues if jobs fail to complete in a timely manner.

This dissertation addresses the problems stated above for data-intensive computing systems. It provides different approaches for both HPC storage systems and big-data storage systems because their differences in principles, architecture, and usage pose distinct challenges. Before studying these systems and addressing their respective problems separately, the discussion of the differences between these two types of systems is established here.

HPC systems are strongly coupled distributed systems, connected by expensive hardware and network links (e.g. InfiniBand [inf]). The application execution principle focuses on *task parallelism*, and thus both its parallel compute processes and I/O requests are tightly coupled and must be executed *together*. This means a failure of any node re-

sults in the failure of the entire application. This is also why the checkpointing I/Os are major sources of I/Os when running such applications, as the periodical save of application progress constitutes much higher amount of data than its original input and final output. Since the defensive I/Os are synchronous, they block the whole application's progress in its defensive I/O phase. The I/Os are served in a remote I/O model, where compute and storage nodes are discrete sets of nodes, and data flows on the network from compute nodes directly to the storage nodes (and vice versa). As discussed above, the increase of HPC system scale reduces Mean Time To Failure (MTTF), adding the need to perform more frequent defensive I/Os. The most widely used programming framework for HPC systems is Message Passing Interface (MPI).

Big-data systems are also distributed systems, while commonly connected by commodity, inexpensive hardware using Ethernet links. The architecture employs *data parallelism*, such that each task is independent of each other for a certain application. Thus, individual tasks of any application can be scheduled *separately*. Currently MapReduce is the most widely accepted computing model in big-data systems. Such systems' fault-tolerance is enabled by restarting any failed task independently on any node, preferably as close to where the data exists as possible. The I/Os are served by a locality model, where computing is shipped to the data location and performs local I/Os there. The replication scheme behind such a storage system results in a much higher probability of executing local I/Os, also making task recovery much easier. The independence of tasks enables each of them to make progress without rolling back the whole application's progress even under failures. Unlike HPC storage systems, big-data storage systems provide a more complex application execution approach, where different phases of a MapReduce application (*map*, *shuffle*, *reduce*) involve local I/Os interleaved with remote I/Os, and persistent I/Os interleaved with intermediate I/Os.

Throughout this dissertation the definition of HPC systems and big-data systems are exclusive of each other, although in reality, both architectures support executing the framework and jobs of the other, blurring the lines between them.

The rest of this chapter presents separately the problems, challenges, and approaches to HPC storage management and big-data storage management.

1.1 HPC Storage Systems

High-performance computing (HPC) systems remain to be indispensable for solving challenging computational problems in many disciplines including science, engineering, medicine, and business. Such systems deliver high performance to applications through parallel computing on large numbers of processors and parallel I/Os on large numbers of storage devices. However, a noteworthy recent trend of HPC is that the applications become increasingly data intensive. On one hand, the emergence of "big-data" is fostering a rapidly growing number of data-driven applications which rely on the processing and analysis of large volumes of data. On the other hand, as applications employ more processors to solve larger and/or harder problems, they are forced to checkpoint more frequently in order to cope with the reduced mean time to failures [OAT+07]. The implication of the proliferation in data-intensive applications in HPC systems is that the I/O performance plays a growing, crucial role in applications on HPC systems.

At the same time, HPC applications are increasingly deployed onto shared computing and storage infrastructures. Similarly to the motivations for cloud computing, consolidation brings significant economical benefits to both HPC users and providers. For data-intensive HPC applications, hosting popular datasets (e.g., human genome, weather data, digital sky survey, Large Hadron Collider experiment data) on shared infrastructure also allows these massive volumes of data to be conveniently and efficiently shared by different applications. Consequently, today's HPC systems are typically not for dedicated use

by particular applications anymore; they are, instead, shared by applications with diverse resource demands and performance requirements.

It is the combination of the above two concurrent trends that makes resource management, particularly the management of shared storage resources an important and challenging problem to HPC systems. Although the processors of an HPC system are relatively easy to partition in a space-sharing manner, the storage bandwidth is difficult to allocate because it has to be time-shared by applications with varying I/O demands. Without proper isolation of competing I/Os, an application's performance may degrade in unpredictable ways when under contention. However, the support of such storage management is generally lacking in HPC systems. In fact, existing HPC storage stack is unable to recognize different applications' I/O workloads—it sees only generic I/O requests arriving from the compute nodes; it is also incapable of satisfying the applications' different storage bandwidth needs—it is often architected to meet the throughput target for the entire HPC system. These inadequacies prevent applications from achieving their desired performance while making efficient use of the HPC resources.

This dissertation presents a new approach to addressing the above limitations and providing application-specific storage bandwidth management through the virtualization of parallel file systems (e.g., Lustre [Lus08], GPFS [SH02], PVFS2 [CLRT00], PanFS [WUA+08]) commonly used in HPC systems. The virtualization framework, named *vPFS*, is able to transparently interpose parallel file system I/Os, differentiate them on a per-application basis, and schedule them according to the applications' performance requirements. Specifically, it is based on 1) the capture of parallel file system data and metadata requests prior to their dispatch to the storage system, 2) distinguishing and queuing of per-application I/Os, and 3) scheduling of queued I/Os based on application-specific bandwidth allocations. vPFS employs a proxy-based virtualization design which enables

the above parallel I/O interposition and scheduling transparently to existing storage systems and applications.

Based on vPFS, various I/O scheduling algorithms can be enabled at the virtualization layer for different storage management objectives. Specifically, this dissertation focuses on proportional sharing schedulers which can provide the much needed control knob for allocating the shared data and metadata services of the parallel storage, much like the allocation of CPU cores in an HPC system. To this end, the dissertation designs a new proportional-share I/O scheduler, SFQ(D)+, which allows applications with diverse I/O sizes and issue rates to share the parallel storage with good application-level fairness and system-level utilization. It recognizes the limitation of the traditional SFQ(D) scheduler by considering the different costs of dispatched I/Os when they are processed by the underlying storage. It further employs the backfilling technique to promote the dispatching of small I/Os and improve the storage resource utilization. This scheduler is employed by the distributed vPFS proxies to provide fair sharing of each individual data and metadata server, while the distributed schedulers also coordinate with one another to achieve fair sharing of the entire parallel storage system's total data and metadata services. To handle the scale of HPC storage system, new threshold- and layout-driven synchronization schemes are invented to support the scheduling coordination with good efficiency.

The vPFS framework can support different parallel file systems and be transparently deployed on existing HPC systems. A prototype of vPFS is implemented by virtualizing PVFS2 [CLRT00], a widely researched and open-source parallel file system, and evaluated with experiments using a comprehensive set of representative HPC benchmarks and applications, including the IOR [SS07] benchmark, BTIO from the NAS Parallel Benchmark suite [BBB+91]), a real-world scientific application WRF [WB05], and a metadata benchmark multi-md-test from PVFS2 [CLRT00]). The results demonstrate that the performance overhead of the proxy-based virtualization and I/O scheduling is small: $< 1\%$

for reads, $< 3\%$ for writes, and $< 3\%$ for metadata accesses in terms of throughput. The results also show that the new SFQ(D)+ scheduler can achieve good proportional sharing for competing applications with diverse I/O patterns: $> 96\%$ of the target sharing ratio for data service and $> 98\%$ for metadata service. It can also provide significantly better performance isolation for an application with small, bursty I/Os (when it is under intensive I/O contention) than the traditional SFQ(D) scheduler (3.35 times better performance) and the native PVFS2 (8.25 times better performance) while still making efficient use of the storage (13.81 times better total throughput than a non-work-conserving scheduler).

In summary, the contributions of this HPC storage management work are as follows:

1. A new virtualization-based parallel storage management approach which is, to the best of our knowledge, the first to allow per-application bandwidth allocation in such an environment without modifying existing HPC systems;
2. The design, implementation, and evaluation of vPFS which is demonstrated experimentally to support low overhead bandwidth management of parallel I/Os as well as metadata operations;
3. Novel distributed SFQ-based scheduling techniques that fit the architecture of HPC parallel storage and support efficient total-service proportional sharing;
4. Adapted backfilling technique specifically for storage systems, to protect small I/Os whose performance is easily undermined by large I/Os;
5. The first experimental evaluation of SFQ-based proportional sharing algorithms in parallel file system environments.

1.2 Big-data Storage Systems

Big-data is an important computing paradigm that becomes increasingly used by many science, engineering, medical, and business disciplines for knowledge discovery, decision making, and other data-driven tasks based on processing and analyzing large volumes of data. These applications are built upon computing paradigms that can effectively express data parallelism and exploit data locality (e.g., MapReduce [DG04]) and storage systems that can provide high scalability and availability (e.g., Google File System [GGL03], Hadoop HDFS [SKRC10]). As the needs of data-intensive computing continue to grow in various disciplines, it becomes increasingly common to use shared infrastructure to run such applications. First, big-data systems often require substantial investments on computing, storage, and networking resources. Therefore, it is more cost-effective for both resource users and providers to use shared infrastructure for big-data applications. Second, hosting popular data sets (e.g., human genome data, weather data, census data) on shared big-data systems allows such massive data to be conveniently and efficiently shared by different applications from different users.

Although computing resources (CPUs) are relatively easy to partition, shared storage resources (I/O bandwidths) are difficult to allocate, particularly for data-intensive applications which compete fiercely for access to large volumes of data on the storage. Existing big-data systems lack the mechanisms to effectively manage shared storage I/O resources, and as a result, applications' performance degrades in unpredictable ways when there is I/O contention. For example, when one typical MapReduce application (WordCount) runs concurrently with a highly I/O-intensive application (TeraGen), WordCount is slowed down by up to 107%, compared to when it runs alone with the same number of CPUs.

I/O performance management is particularly challenging for big-data systems because of two important reasons. First, big-data applications have complex I/O phases (e.g., rounds of map and reduce tasks with different amounts of inputs, intermediate results, and outputs for a MapReduce application), which makes it difficult to understand their I/O demands and allocate I/O resources properly to meet their performance requirements. Second, a big-data application is highly distributed across many datanodes, which makes it difficult to coordinate the resource allocations across all the involved nodes needed by the data-parallel application. For example, the performance of a MapReduce application depends on the received total storage bandwidth from all the nodes assigned to its map and reduce tasks.

This dissertation presents *IBIS*, an Interposed Big-data I/O Scheduler, to provide performance differentiation for competing applications' I/Os in a shared big-data system. This scheduler is designed to address the above-mentioned two challenges. First, *how to effectively differentiate I/Os from competing applications and allocate the shared storage bandwidth on the individual nodes of a big-data system?* IBIS introduces a new I/O interposition layer upon the distributed file system in a big-data system, and is able to transparently intercept the I/Os from the various phases of applications and isolate and schedule them on every datanode of the system. IBIS also employs a new proportional-share I/O scheduler, SFQ(D2), which can automatically adapt I/O concurrency based on the storage load and achieve strong performance isolation with good resource utilization. Second, *how to efficiently coordinate the distributed I/O schedulers across datanodes and allocate the big-data system's total I/O service to the data-parallel applications?* IBIS provides a scalable coordination scheme for the distributed SFQ(D2) schedulers to efficiently coordinate their scheduling across the datanodes. The schedulers then adjust their local I/O scheduling based on the global I/O service distribution and allow the applications to proportionally share the entire system's total I/O service.

The IBIS prototype is implemented in Hadoop/YARN, a widely used big-data system, by interposing HDFS as well as the related local and network I/Os transparently to the applications, and it is able to support the I/O management of diverse applications from different big-data frameworks. It is evaluated using a variety of representative big-data applications (WordCount, TeraSort, TeraGen, Facebook2009 [SWI], TPC-H queries on Hive [TSJ+10]). The results confirm that IBIS can effectively achieve total-service proportional bandwidth sharing for diverse applications in the system. They also show that IBIS can support various important performance polices. It achieves strong *performance isolation* for a less I/O-intensive workload (WordCount, Facebook2009, TPC-H) when under heavy contention from a highly I/O-intensive application (TeraGen and TeraSort), which outperforms native Hadoop by 99% for WordCount and 15% for TPC-H queries. This result is accomplished while still allowing the competing application to make good progress and to fully utilize the storage bandwidth ($< 4\%$ reduction in total throughput). IBIS can also achieve excellent *proportional slowdown* for competing applications (TeraSort vs. TeraGen) and outperforms native Hadoop by 30%. Finally, the use of IBIS introduces small overhead in terms of both application runtime and resource usages.

Overall, unlike most of the related works which focus on improving the I/O efficiency of big-data systems [AGW+12, GRZ13], this dissertation addresses the problem of I/O interference and performance management in big-data systems, which is not adequately addressed in the literature. Although existing mechanisms such as cgroups [con] can be employed to manage the contention among local I/Os, as the results in this dissertation will show, they are insufficient due to the lack of control on distributed I/Os which are unavoidable for big-data applications. IBIS therefore complements the existing solutions for CPU and memory management of big-data systems, and provides the missing control knob for I/O management which is much needed by increasingly data-intensive applications. Compared to the few related works [WVA+12b, SFS12, WVA+12a] that also

studied the performance management of big-data storage, IBIS supports applications that are more challenging (with complex computing and I/O demands) and diverse (including both batch and query workloads). In summary, this dissertation makes the following contributions in big-data storage management:

1. A thorough study of the performance interference among competing big-data applications caused by I/O contention on the shared storage;
2. A new interposed big-data I/O scheduling framework which can transparently schedule the different types of I/Os from a MapReduce application and efficiently coordinate the scheduling across the distributed storage nodes to enforce the policy for total-service sharing of the entire system;
3. A new adaptive proportional-share I/O scheduling algorithm (SFQ(D2)) which can automatically optimize the tradeoff between fair sharing and resource utilization and provide strong performance differentiation to applications with high utilization of the system;
4. A prototype built upon widely used Hadoop MapReduce system, which is shown experimentally to deliver effective total-service proportional sharing and support flexible performance polices including performance isolation and proportional slowdown.

1.3 Outline

The outline of the dissertation is as follows: Chapter 2 discusses the background and related work in HPC and big-data storage management; Chapter 3 and Chapter 4 present the performance management of HPC and big-data storage systems, respectively; and Chapter 5 concludes the dissertation and outlines the future work.

CHAPTER 2

RELATED WORK

2.1 HPC Storage Systems

In a typical HPC system, applications running on the compute nodes access their data stored on the storage nodes through a parallel file system. A parallel file system (e.g., Lustre [Lus08], GPFS [SH02], PVFS2 [CLRT00], PanFS [WUA+08]) consists of clients, data servers, and metadata servers. The clients run on the compute nodes (or dedicated intermediate I/O nodes) and provide the interface (through POSIX or MPI-IO [TGL96]) to the parallel file system. Metadata servers are responsible for managing file naming, data location, and file locking. Data access typically has to first go through a metadata server to obtain the appropriate permission and the location on the corresponding data servers for the requested data. To avoid the metadata server from becoming a bottleneck, parallel file systems can distribute metadata management across several metadata servers. Finally, data servers run on the storage nodes and are responsible for performing reads and writes on the locally stored application data. Each data request issued by a client is usually striped across multiple data servers to achieve high performance by serving the striped requests in parallel on their storage nodes.

In current HPC systems (Figure 2.1), the storage infrastructure is considered as opaque by applications: it is shared by all the compute nodes and it serves applications' I/O demands in a best-effort manner. Although it is straightforward to partition the compute nodes (and their processors) among multiple concurrent applications, the parallel file system storage has to serve the concurrent I/O requests from all the applications that are running in the system, which often have distinct I/O access patterns and performance requirements. However, the parallel file system based storage is not designed to recognize the different I/O demands from applications—it only sees generic I/O requests arriving

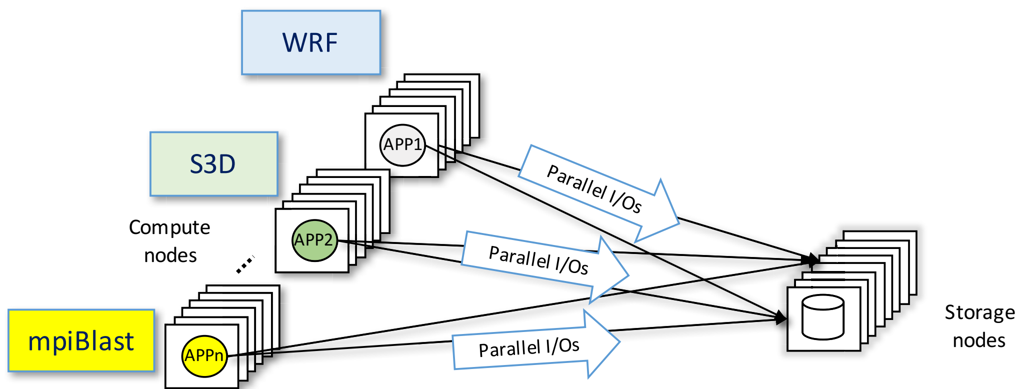


Figure 2.1: The architecture of a parallel storage system

from the compute nodes. Neither is the storage system designed to satisfy the different performance requirements from applications—it is engineered to meet the maximum throughput target for the entire HPC system.

2.2 Big-data Storage Systems

Typical big-data computing systems are often built upon a highly scalable and available distributed file system. In particular, Google File System (GFS) [SEP+97] and its open-source clone Hadoop Distributed File System (HDFS) [SKRC10] provide storage for massive amounts of data on a large number of nodes built with inexpensive commodity hardware while providing fault tolerance at scale. Big-data applications then build upon the I/O interface provided by such a distributed file system and are executed on the nodes in a data-parallel fashion. In particular, the MapReduce programming model and associated run-time systems are able to automatically execute user-specified map and reduce functions in parallel, and handle job scheduling and fault tolerance [DG04]. Higher-level storage services such as databases (e.g. Bigtable [CDG+06], HBase [Hba], Spanner [CDE+12]) can be futher built upon these distributed file systems. Therefore,

this dissertation focuses on big-data storage systems of the GFS/HDFS kind. While big-data databases can also be built directly on raw storage without relying on a distributed file system [DHJ+07, Vol, LM10, Klo10], they are not the focus of this dissertation.

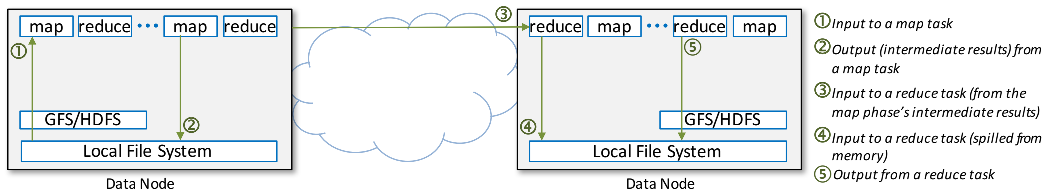


Figure 2.2: The architecture of a big-data storage system

Both the map and reduce phases of a MapReduce application can spawn large numbers of map and reduce tasks on the GFS/HDFS nodes to process data in parallel. They often have complex but well-defined I/O phases (Figure 2.2). A *map* task is preferably scheduled to the node where its input data is stored. It reads the input from GFS/HDFS (either via the local file system or across the network) and *spills* and *merges* key-value pairs onto the local file system as intermediate results. A *reduce* task starts by *copying/shuffling* its inputs from all the map tasks' intermediate results (either stored locally or across the network). It then *merges* the copied inputs, performs the *reduce* processing, and generates final output to GFS/HDFS. Each of the above phases can have different bandwidth demands in the input and output. Moreover, given the same volume of data to a map or reduce task, it can take different amounts of time to process the data depending on the application's computational complexity.

2.3 Related Storage Management Work

2.3.1 HPC Storage Management

Storage resource management has been studied in related work in order to service competing I/O workloads and meet their desired throughput and latency goals. Such management can be embedded in the shared storage resource's internal scheduler (e.g., disk schedulers) (Cello [SV98], Stonehenge [HPC04], YFQ [BBG+99], PVFS [RI01]), which has direct control over the resource but requires the internal scheduler to be accessible and modifiable. The management can also be implemented via virtualization by interposing a layer between clients and their shared storage resources (Façade [LMA03], SLEDS [CAP+03], SFQ(D) [JCK04], GVFS [ZZF06]). This approach does not need any knowledge of the storage resource's internals or any changes to its implementation. vPFS follows this approach in order to support existing HPC setups and diverse parallel file systems. Although this virtualization approach has been studied for different storage systems, to the best of our knowledge, vPFS is the first to study its application on parallel file system based storage systems. Moreover, vPFS embodies new designs that address the unique characteristics and requirements of such systems.

Various scheduling algorithms have been investigated in related storage management solutions. They employ techniques such as virtual clocks, leaky buckets, and credits for proportional sharing [JCK04] [HPC04] [ZSW+06] [GMV10], earliest-deadline first (EDF) scheduling to guarantee latency bounds [LMA03], feedback-control with request rate throttling [KKZ05], adaptive control of request queue lengths based on latency measurements [GAW09], and scheduling of multi-layer storage resources based on online modeling [SLG+09]. The effectiveness of these scheduling algorithms is unknown for an HPC parallel storage system. This gap can be mainly attributed to the lack of support for per-application bandwidth allocation in such a system. The vPFS framework bridges

this gap by enabling various parallel I/O schedulers to be instantiated without imposing changes to parallel file system clients and servers. It is also the first to study proportional sharing algorithms on a parallel file system.

The majority of the storage resource schedulers in the literature focuses on the allocation of a single storage resource (e.g., a storage server, device, or a cluster of interchangeable storage resources) and addresses the local throughput or latency objectives. LexAS [GV05] was proposed for fair bandwidth scheduling on a storage system with parallel disks, but I/Os are not striped and the scheduling is done with a centralized controller. DSFQ [WM07] is a distributed algorithm that can realize total service proportional sharing across all the storage resources that satisfy workload requests. However, it faces challenges of efficient global scheduling when applied to an HPC parallel storage system, which are addressed by the vPFS and its enabled distributed parallel I/O scheduling.

U-Shape [ZDJ11] is a related project that tries to achieve application-desired performance by first profiling the application's instantaneous throughput demands and then at runtime scheduling the application's I/Os according to the predicted demands. However, it does not address the often unpredictable contentions in a real-world HPC system where applications with complex behaviors compete on the shared parallel storage system in a convolved manner. In comparison, vPFS provides proportional sharing of the data and metadata services for diverse applications without assuming *a priori* knowledge of the applications. It offers the key missing control knob in HPC management which can be utilized to realize different high-level performance policies such as proportional slowdown and performance isolation.

Recognizing the importance of metadata management, HPC researchers proposed various solutions to improve metadata access performance. [MBL+11] and [PG11] studied directory distribution for highly scalable parallel file systems. [AZ11] proposed parallel file system delegation techniques to offload the management of parallel storage space

to applications, relieving the metadata management bottleneck at the parallel file system. [CLR+09] considered techniques to improve metadata access efficiency in PVFS by precreating objects and batching requests. [WUA+08] studied the use of SSD-based buffers for accelerating metadata accesses. These solutions are complementary to vPFS in that vPFS can benefit from of the metadata performance enhancements made by these related solutions while providing proportional sharing of the metadata service which they lack.

Finally, there is related work that also adopts the approach of adding a layer upon an existing parallel file system deployment in order to extend its functionality or improve its performance (pNFS [HH05], PLFS [BCG+10], ZOID [IRYB08]). In addition, cross-server coordination has also been considered in the related work [ZDJ10] [SYS+11] to improve spatial locality and I/O performance. These efforts do not address the fair sharing of a parallel file system by concurrent applications and are hence also complementary to this dissertation's vPFS-based solutions.

2.3.2 Big-data Storage Management

Existing big-data systems offer simple core resource management functions. Hadoop MapReduce [had] allocates CPU resources in terms of *slots* to map or reduce tasks, where the number of available slots is set according to the number of CPU cores in the system. Recent developments such as Mesos [HKZ+11] and YARN [VMD+13] allow the allocation of both CPU and memory resources to competing big-data applications. But the management of shared I/O bandwidth among competing applications is missing from existing systems, which is crucial to the performance of inherently I/O intensive big-data applications.

In an HPC system as well as others, the computing infrastructure consisting of large numbers of compute nodes is often the core of the system. A parallel/distributed file system uses a much smaller number of storage nodes to provide remote data access to applications executed on the compute nodes. In contrast, in a big-data system, *the storage infrastructure is the core of the system*. Computing tasks are distributed to large numbers of storage nodes to process the stored data. Moreover, although traditional MPI-based HPC applications can also be I/O demanding, their I/O operations are mostly generated by large, sequential checkpointing events. In contrast, big-data applications have intertwined I/O phases which generate both reads and writes to storage either locally or across the network. Consequently, the scale and complexity of a big-data system are substantially higher than an HPC system, and the I/O demands of big-data applications are also substantially more challenging to manage. This dissertation recognizes these unique challenges and addresses them with new solutions upon the general interposition principles.

To the best of our knowledge, the problem of I/O management for MapReduce-type big-data systems is largely unexplored in the literature. For other types of big-data systems, Frosting [WVA+12b] provides a scheduling layer upon HBase which dynamically controls the number of outstanding requests and proportionally shares the HBase storage among competing clients. However, it treats the entire HBase storage stack as a single black box and is agnostic to how the competing I/Os are distributed to the underlying storage nodes. Consequently, it is difficult to achieve good performance isolation when I/Os compete on the individual shared nodes; while in order to provide any performance guarantee, it has to throttle HBase I/Os as long as one of the nodes is overloaded, leaving the others underutilized. In contrast, IBIS manages I/Os at the lower GFS/HDFS layer and in a distributed manner. This approach can provide more effective I/O performance differentiation while making efficient use of the underlying storage resources.

PISCES [SFS12] provides fair sharing of key-value storage by controlling requests dispatched to storage nodes according to the shares. In comparison, in a MapReduce-type big-data system an application's task distribution is driven by both CPU slot requirement and data locality, and its I/O demands are much more complex—including multiple phases of local and network I/Os, and diverse—with different intensities on the various types of I/Os. Hence, I/O management in a MapReduce system cannot be achieved by merely controlling task dispatching, and has to rely on both local I/O scheduling and global coordination which are part of the IBIS solution.

Cake [WVA+12a] presents a two-level scheduling approach to meeting the performance goal of latency-sensitive applications (HBase) when they are consolidated with throughput-oriented applications (MapReduce), but it cannot provide any performance guarantee to the latter. In comparison, IBIS supports both types of applications.

Several works studied other orthogonal aspects of big-data storage: PACMan [AGW+12] manages memory-based caching of map task inputs to improve application performance; iShuffle [GRZ13] improves the performance of intermediate I/Os. However, they would still need a storage management solution like IBIS to provide performance isolation for the intermediate I/Os and the I/Os that trickle down the memory cache layer among concurrent applications.

2.3.3 Management of Other Types of Storage Systems

I/O interposition is a technique often used to manage a shared storage resource that does not provide native knobs for controlling its competing I/Os. It has been employed in the related work to realize a proportional bandwidth scheduler for a shared file service [JCK04], to create application-customized virtual file systems upon a shared network file system [ZF06], and to manage the performance of a parallel file system based

storage system [XAZ+12]. Big-data systems present unique challenges to I/O management because of the complexity (different types of I/Os), diversity (different levels of intensity), and scale (many datanodes) of the I/O contention. These are addressed by the techniques embodied in IBIS, including holistic interposition of HDFS, local file system, and network I/Os, an adaptive proportion-share I/O scheduler, and scalable coordination of distributed I/O scheduling.

There are also related works on the performance management of other types of storage systems. Horizon [PSB10] can provide global minimum throughput guarantee for a RAID storage system, but it requires a centralized controller to assign deadlines to requests, which is difficult to apply to a big-data system where I/Os are issued directly by local tasks on each datanode. A two-level scheduler [ZSW+06] was proposed for meeting I/O latency and throughput targets, but it supports only local I/O scheduling, whereas a big-data system requires the distributed storage management provided by IBIS.

CHAPTER 3

HPC STORAGE SYSTEMS MANAGEMENT

3.1 Introduction

High-performance computing (HPC) systems remain to be indispensable for solving challenging computational problems in many disciplines including science, engineering, medicine, and business. Such systems deliver high performance to applications through parallel computing on large numbers of processors and parallel I/Os on large numbers of storage devices. However, a noteworthy recent trend of HPC is that the applications become increasingly data intensive. On one hand, the emergence of “big data” is fostering a rapidly growing number of data-driven applications which rely on the processing and analysis of large volumes of data. On the other hand, as applications employ more processors to solve larger and/or harder problems, they are forced to checkpoint more frequently in order to cope with the reduced mean time to failures [OAT+07]. The implication of the proliferation in data-intensive applications in HPC is that the I/O performance plays an growing, crucial role in applications on HPC systems.

At the same time, HPC applications are increasingly deployed onto shared computing and storage infrastructures. Similarly to the motivations for cloud computing, consolidation brings significant economical benefits to both HPC users and providers. For data-intensive HPC applications, hosting popular datasets (e.g., human genome, weather data, digital sky survey, Large Hadron Collider experiment data) on shared infrastructure also allow these massive volumes of data to be conveniently and efficiently shared by different applications. Consequently, today’s HPC systems are typically not for dedicated use by particular applications any more; they are, instead, shared by applications with diverse resource demands and performance requirements.

It is the combination of the above two concurrent trends that makes resource management, particularly the management of shared storage resources an important and challenging problem to HPC systems. Although the processors of an HPC system are relatively easy to partition in a space-sharing manner, the storage bandwidth is difficult to allocate because it has to be time-shared by applications with varying I/O demands. Without proper isolation of competing I/Os, an application's performance may degrade in unpredictable ways when under contention. However, the support of such storage management is generally lacking in HPC systems. In fact, existing HPC storage stack is unable to recognize different applications' I/O workloads—it sees only generic I/O requests arriving from the compute nodes; it is also incapable of satisfying the applications' different storage bandwidth needs—it is often architected to meet the throughput target for the entire HPC system. These inadequacies prevent applications from achieving their desired performance while making efficient use of the HPC resources.

Virtual PFS is the new approach to addressing the above limitations and providing application-specific storage bandwidth management through the virtualization of parallel file systems (e.g., Lustre [Lus08], GPFS [SH02], PVFS2 [CLRT00], PanFS [WUA+08]) commonly used in HPC systems. The virtualization framework, named *vPFS*, is able to transparently interpose parallel file system I/Os, differentiate them on a per-application basis, and schedule them according to the applications' performance requirements. Specifically, it is based on 1) the capture of parallel file system data and metadata requests prior to their dispatch to the storage system, 2) distinguishing and queuing of per-application I/Os, and 3) scheduling of queued I/Os based on application-specific bandwidth allocations. vPFS employs a proxy-based virtualization design which enables the above parallel I/O interposition and scheduling transparently to existing storage systems and applications.

The rest of this chapter is organized as follows: Section 3.2 shows the need for HPC storage systems management with an example; Section 3.3 presents the design of a virtualized parallel file system layer; Section 3.4 describes a novel algrorithm to cope with unfairness of I/O requests in the storage; Section 3.5 introduces a scalable decentralized synchronization scheme for parallel file system schedulers; Section 3.6 discusses the need and method for scheduling of metadata I/Os; and Section 3.7 presents the evaluation of the vPFS solution on PVFS for overhead, effectiveness and efficiency.

3.2 Motivation

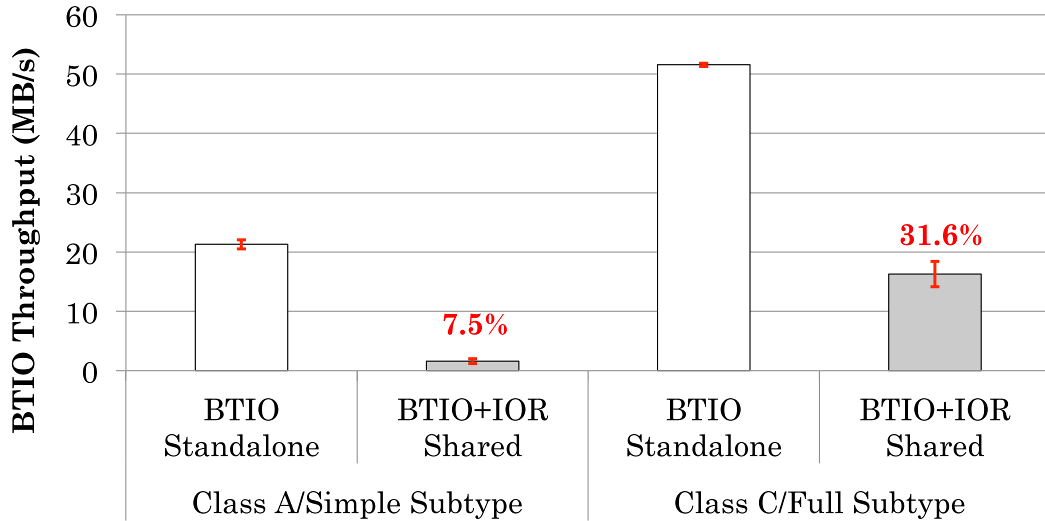
In a typical HPC system, applications running on the compute nodes access their data stored on the storage nodes through a parallel file system. A parallel file system (e.g., Lustre [Lus08], GPFS [SH02], PVFS2 [CLRT00], PanFS [WUA+08]) consists of clients, data servers, and metadata servers. The clients run on the compute nodes (or dedicated intermediate I/O nodes) and provide the interface (through POSIX or MPI-IO [TGL96]) to the parallel file system. Metadata servers are responsible for managing file naming, data location, and file locking. Data access typically has to first go through a metadata server to obtain the appropriate permission and the location on the corresponding data servers for the requested data. To avoid the metadata server from becoming a bottleneck, parallel file systems can distribute metadata management across several metadata servers. Finally, data servers run on the storage nodes and are responsible for performing reads and writes on the locally stored application data. Each data request issued by a client is usually striped across multiple data servers to achieve high performance by serving the striped requests in parallel on their storage nodes.

In current HPC systems, the storage infrastructure is considered as opaque by applications: it is shared by all the compute nodes and it serves applications' I/O demands in

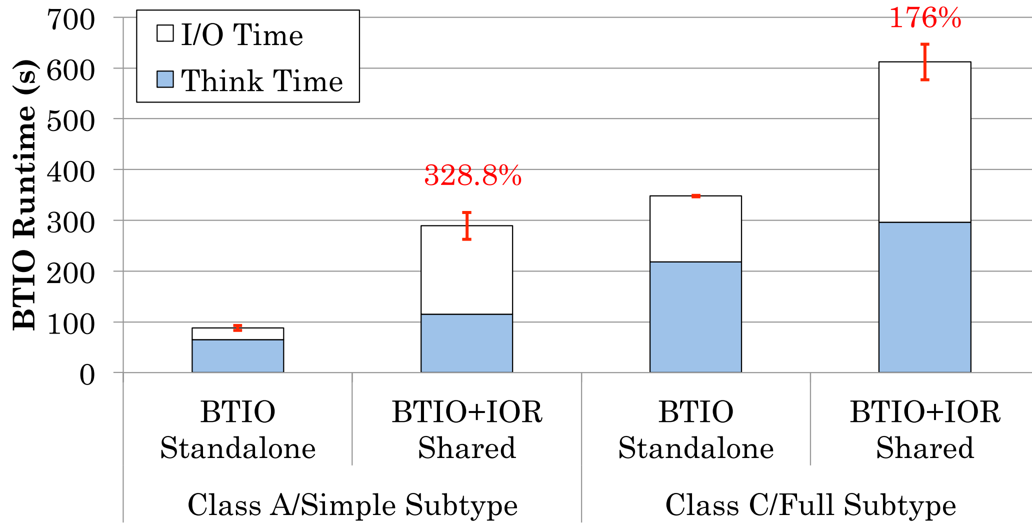
a best-effort manner. Although it is straightforward to partition the compute nodes (and their processors) among multiple concurrent applications, the parallel file system storage has to serve the concurrent I/O requests from all the applications that are running in the system, which often have distinct I/O access patterns and performance requirements. However, the parallel file system based storage is not designed to recognize the different I/O demands from applications—it only sees generic I/O requests arriving from the compute nodes. Neither is the storage system designed to satisfy the different performance requirements from applications—it is engineered to meet the maximum throughput target for the entire HPC system.

HPC applications in fact differ by several orders of magnitude in storage bandwidth requirements. For example, WRF [WB05] may require hundreds of Megabytes of inputs and outputs at the beginning and end of its execution; mpiBLAST [DCF03] may load Gigabytes of genome data when it starts but not much I/Os afterwards; S3D [SHC+06] can produce Gigabytes of restart files periodically in order to tolerate failures during its execution. Applications also have different priorities, e.g., due to different levels of urgency or business value, which should be reflected on the scheduling of their I/Os. For example, the execution of WRF for the forecast of an incoming hurricane should be given the highest priority, but it may not be necessary to dedicate the entire system to WRF, as the obtainable speedup often does not scale to the system size. Therefore, applications with different storage bandwidth demands and performance requirements are multiplexed on the shared storage system, which will become increasingly more common with the continued scale-up of HPC systems. Hence, per-application allocation of shared parallel storage bandwidth is key to delivering application desired performance, which is generally lacking in existing HPC systems.

As a motivating example, Figure 3.1a shows the impact of I/O contention on a typical parallel storage system shared by two applications represented by IOR [SS07], which is-



(a) BTIO's throughput



(b) BTIO's runtime

Figure 3.1: BTIO performance slowdown

sues continuous checkpointing I/Os and NPB BTIO [BBB+91], which generates outputs interleaved with computation. Each application has a separate set of compute nodes but they compete for the same set of parallel file system servers. The figure compares the performance of BTIO between when it runs alone (*Standalone*) without any I/O contention, and when it runs concurrently with IOR (*Shared*). The chosen experiment workloads are *Class C* with *full* subtype (using collective I/O [TGL99]) and *Class A* with *simple* subtype (without collective I/O).

The expectation from the BTIO user's perspective when under I/O contention is either no impact (no loss in I/O throughput and no increase in runtime) or at least fair sharing (50% loss in I/O throughput and 100% increase in I/O time). However, the results in Figure 3.1b show that BTIO suffers much more severe performance drop even though the two applications are using separate compute nodes. For Class A with smaller I/Os, the I/O time is increased by 657% (Figure 3.1b) and the I/O throughput is reduced by 92.5% (Figure 3.1a); and as a result the total runtime is increased by 228.8%. For Class C with larger I/Os, the performance loss is relatively smaller: the I/O time is still increased by 143.5%, and the I/O throughput is reduced by 68.4%; and as a result, the total runtime is increased by 76%. (More details on this experiment are presented in Section 3.7.) These results demonstrate the need for bandwidth management in a parallel storage system, a problem that becomes increasingly pronounced as HPC systems grow in size. It is desirable to provide fair bandwidth sharing so that each application achieves predictable performance regardless of the contention from other applications in a shared HPC system.

A possible solution to this problem is to statically allocate parallel file system servers to each application, but data cannot be easily swapped in and out as processes do, so this solution is not feasible in HPC systems with a large number and dynamic sets of applications. This approach also disables the sharing of common data among different applications. Some systems limit the number of parallel file system clients that an appli-

cation can access [NLC08] [YSH$^+$06]. This approach allows an application to always get some bandwidth through its allocated clients. But as shown in the above experiment, it cannot support strong bandwidth isolation because the parallel file system servers are still shared by all the applications without any isolation.

3.3 Parallel Storage System Virtualization

The general strategy taken by vPFS to enable I/O scheduling for HPC parallel storage systems is based on the *virtualization principles*, where an indirection layer exposes the parallel file system interfaces already in use by the storage system for I/O accesses. This strategy allows applications to time-share the I/O resources without modifications, while enforcing performance isolation among them. Parallel I/O schedulers can be layered upon this virtualization layer to address the limitations of existing parallel file systems discussed in Section 2.3.1 without changing the interface exposed to applications. To create this virtualization layer, vPFS takes a *user-level proxy-based I/O interposition* approach by inserting a layer of parallel I/O proxies between the shared native parallel file system clients and servers, which differentiates per-application I/Os and enforces their resource allocation. This proxy-based virtualization approach can be applied transparently to existing HPC system deployments, while the experimental results in Chapter 3.7 show that the overhead introduced by the user-level proxies is small. It can support different parallel file system protocols as long as the proxy understands the protocols and handles the I/Os accordingly.

Specifically in vPFS, the parallel I/O proxies are spawned on every parallel file system server to broker the application's I/Os across the system, where the requests issued by a parallel file system client are first processed and queued by a proxy and then forwarded to the native parallel file system server for the actual data access (Figure 3.2). To

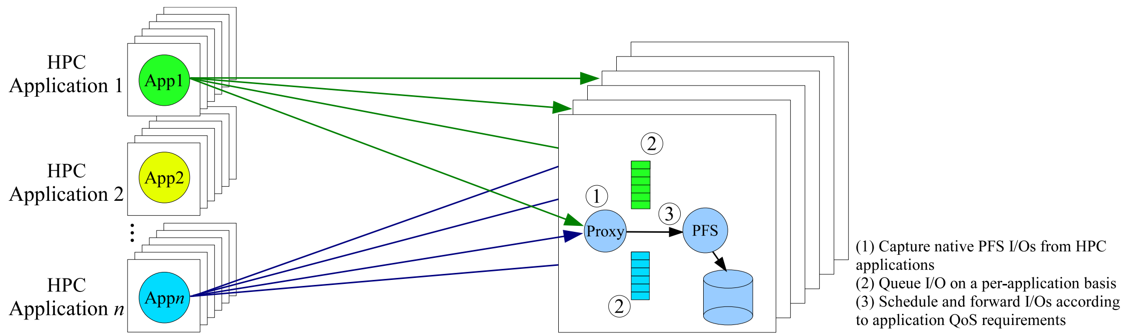


Figure 3.2: The architecture of PFS virtualization

differentiate the I/Os from different applications, a proxy can use their source network addresses (IPs) to identify their ownership, because HPC systems commonly partition compute nodes across concurrent applications so each node executes a single application's processes. In the case when multiple applications are run on the same compute node, each application's I/Os can be directed to a specific port of the proxy so that its I/O can be uniquely identified using the source network address and port number combination.

The placement of vPFS proxies does not have to be restricted to the native parallel file system servers in order to create virtual parallel file systems. They can in fact be placed anywhere along the data path between the clients and servers. For example, the proxies can run on a subset of the compute nodes, which are dedicated to provide I/O services for the rest of the compute nodes. Such a proxy placement can be leveraged to deploy vPFS in HPC systems that do not allow any third-party program on the data servers. It is also useful when the network bandwidth between the compute and I/O nodes is a bottleneck and needs to be allocated on a per-application basis.

Although such a virtualization approach has been applied to the scheduling of centralized storage servers [ZF06, JCK04], vPFS is the first to take this approach to manage a highly distributed storage system, and consequently it needs to address the scalability challenges in the management. In an HPC system, it is infeasible to rely on a single proxy

to decide the scheduling of I/Os (as the previous work does) for the entire distributed storage system, although such an approach would be convenient to enforce global bandwidth allocations. Instead, vPFS is designed with decentralized proxies which cooperate on I/O scheduling to collectively deliver global bandwidth allocations, and it considers new techniques to reduce the overhead from cross-proxy communication for global scheduling synchronization. A variety of parallel I/O scheduling algorithms with different objectives can be deployed at the vPFS virtualization layer by using the proxies to monitor and control I/O executions. This dissertation focuses on proportional sharing and address the challenges of realizing such scheduling efficiently in large-scale parallel storage systems.

3.4 Fair Share of Parallel Storage I/Os

Proportional sharing of parallel storage bandwidth is important to HPC applications because their performance targets are often specified in terms of turnaround times which depend on the shares of CPU cycles and I/O bandwidth that the applications can get from the system. It is relatively straightforward to allocate CPU shares based on cores and time slices, but it is non-trivial to allocate I/O bandwidth shares. Proportional-share I/O schedulers can provide this key missing control knob in HPC resource management. Proportional resource sharing is defined as when the total demand is greater than available resource, each application should gets a *share* of the resource proportionally to its assigned *weight* [KMT98, MW00]. In an HPC system, the weights can be set based on the high-level policies, such as application priorities, and the proportional-share scheduler would enforce such policies among the applications that share the parallel file system's I/O bandwidth. For example, if two applications are assigned weights 4 and 1, then their shares of the system's I/O bandwidth should be 80% and 20%, respectively, whenever their combined demand exceeds the total available bandwidth. Because only the relative

values of the weights matter to the bandwidth allocation, in the dissertation, the weight assignment to the applications is often specified in terms of the ratio among the weights.

3.4.1 Review of SFQ and SFQ(D)

The proportional-share scheduler is built upon the SFQ family of schedulers because of their computational efficiency, work conserving nature, and theoretical provable fairness. To help the readers understand the new proportional-share scheduler, a brief summary of SFQ and SFQ(D) is provided here. In essence, SFQ schedules the backlogged requests from different applications using a priority queue, where each request's priority is positively affected by its application's weight and negatively affected by its cost (often estimated based on the size of the request). The scheduler can dispatch only one outstanding request, and it decides which one to dispatch based on the virtual *start time* and *finish time* of the requests in its queue. A new request's start time is generally assigned based on the same application's previous finish time, and its finish time equals to its start time plus its cost. Therefore, when the scheduler dispatches the queued requests according to the increasing order of their start times, each competing application is able to get a fair share of service on the shared resource. When applications are assigned different weights (e.g., based on their priorities), these weights are used to adjust the finish times—the cost of a request reduces proportionally to its application's weight—so that the applications receive service proportionally to their weights.

The scheduler is work-conserving in that when an application does not have enough requests to use up its allocated share of service, the scheduler does not wait for the application and instead immediately dispatches the next request in the queue with the smallest start time. But when this application's next request comes, its start time is set to the current global virtual time—which is always advanced by the scheduler based on the start

time of its last dispatched request—instead of the application's previous request's finish time, so the service that it did not use due to previous idleness is forfeited.

Based on the above basic SFQ algorithm, the SFQ(D) [JCK04] algorithm is designed for proportional sharing of storage resources that are commonly able to handle multiple outstanding requests concurrently. The level of concurrency that the shared storage resource supports is captured by the parameter $\mathscr{D}$ in SFQ(D). The scheduler follows the original SFQ algorithm when assigning timestamps and dispatching queued requests according to the increasing order of their start times, but it allows up to $\mathscr{D}$ outstanding I/Os to be serviced concurrently by the underlying storage in order to take advantage of the available concurrency of the resource.

The choice of $\mathscr{D}$ has important implications on both fairness and resource utilization in a real system. Theoretical bound of fairness comes from the assumption that all applications have backlogged requests in the scheduler. However, it is not the case in reality. On one hand, a larger $\mathscr{D}$ allows more concurrent I/Os and a higher utilization of the storage, but it may hurt fairness because it allows a more aggressive workload to dispatch more I/Os while a less aggressive one to lose more share due to the work-conserving nature of the scheduler. As the dispatched I/Os are out of the control of the scheduler, they may also overload the storage and cause significant delays to the following I/Os from other workloads. On the other hand, a smaller $\mathscr{D}$ gives the scheduler a tighter control on the amount of I/O share that a more aggressive workload can steal from others, and allows the less aggressive workloads to establish queued I/Os for the scheduler to choose from when it is ready to dispatch more. It can thus improve fairness among the competing workloads but may lead to underutilization of the storage.

Therefore, it is important to strike a balance between fairness and utilization with an optimal $\mathscr{D}$ under mixed I/O workloads, which is a challenging problem addressed by vPFS. The inherent limitation of SFQ(D) is that $\mathscr{D}$ does not capture the impact from

the size of each I/O. Every dispatched I/O occupies one out of the $\mathscr{D}$ slots, regardless of the size of the I/O, although a larger I/O generates a higher load on the storage than a smaller one. This limitation becomes pronounced when applying SFQ(D) to an HPC parallel storage system, which has been traditionally oriented to service large, sequential I/Os (e.g., checkpointing) but is also seeing increasingly more small, random I/Os (e.g., visualization) [CLR+09]. The BTIO motivation example in Figure 3.1 shows that small I/Os are vulnerable under the contention of large I/Os (the Class A workload experiences 152% more increase in runtime and 400% more slowdown in throughput than the Class C workload). The new scheduler addresses this limitation of SFQ(D) and provides fair scheduling of both large and small parallel I/Os upon vPFS.

3.4.2 Variable Cost I/O Depth Allocation

The new SFQ(D)+ scheduler recognizes the different costs of outstanding requests in the underlying storage by allocating different number of *slots* of the total I/O depth to them based on their slot cost $\mathscr{L}$. Once $\mathscr{D}$ is chosen, different sized I/Os use different number of slots out of $\mathscr{D}$, and the total slot cost of all outstanding I/Os should not exceed $\mathscr{D}$. The use of the variable slot costs in the algorithm is described in pesudocode in Algorithm 1. This enhancement to the original SFQ(D) algorithm can effectively protect small, low I/O rate workloads and provide better fairness when they are contended by large, intensive workloads. Small I/Os are affected by less large I/Os when they are dispatched together. Low I/O rate I/Os also wait less for the outstanding large I/Os to complete. SFQ(D)+ takes control of the contention among the outstanding I/Os in the underlying storage before they are dispatched and get out of the hand of the scheduler. I/Os are differentiated not only by their relative order in the scheduler's queue, per the original SFQ(D), but also by their *variable slot cost* $\mathscr{L}$ in the shared storage. A key question to the application of SFQ(D)+

is that how to determine the variable slot cost $\mathscr{L}$ of requests in terms of their use of the underlying storage's I/O depth. It is addressed by profiling the latencies of I/Os of various sizes on the storage. Because I/Os of the same size often require the same of amount of processing and incur similar latencies in the storage, they can be considered to have the same slot cost and their latency can be used to estimate the value of $\mathscr{L}$. This methodology is consistent with the common use of I/O size as the estimate of I/O cost in assigning finish tags in SFQ. Note that for storage where read and write requests incur different costs (e.g., flash storage), different depth costs can be used for reads and writes.

In order to apply this new SFQ(D)+ algorithm to scheduling a shared storage resource, there are two key parameters that need to be determined: first, the $\mathscr{D}$ parameter which represents the number of outstanding I/Os that the storage can handle; and second, the $\mathscr{L}$ parameter which represents the slot cost of an outstanding I/O. Note that in principle these two parameters need to be determined for each major type of I/Os serviced by the storage. For parallel file system based storage, a single $\mathscr{D}$ and a single $\mathscr{L}$ value are sufficient to differentiate the large and small I/Os in such a system, which can be determined by profiling the storage as discussed in the rest of this section.

First, the *throughput model* that captures the relationship between the I/O throughput and the number of outstanding I/Os is built to determine the $\mathscr{D}$ parameter of the storage system. The traditional SFQ(D) algorithm [JCK04] also requires profiling to determine $\mathscr{D}$, but it does not consider the dependence of this parameter to the I/O sizes. In SFQ(D)+, the profiling focuses on building a throughput model for small I/Os that the storage serves, so that the cost of large I/Os can be expressed as multiple depth slots where each slot represents the cost of processing a small I/O. In such a throughput model, the throughput generally increases with the growth of the number of outstanding I/Os, and flattens out when the storage becomes saturated. The number of outstanding I/Os that saturates the storage determines the $\mathscr{D}$ parameter for this particular I/O size.

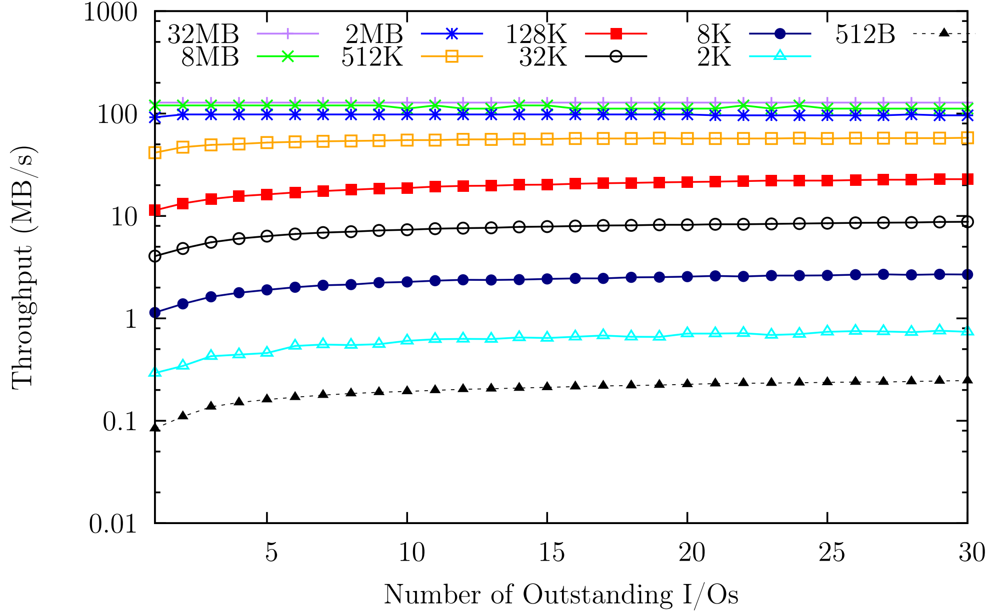
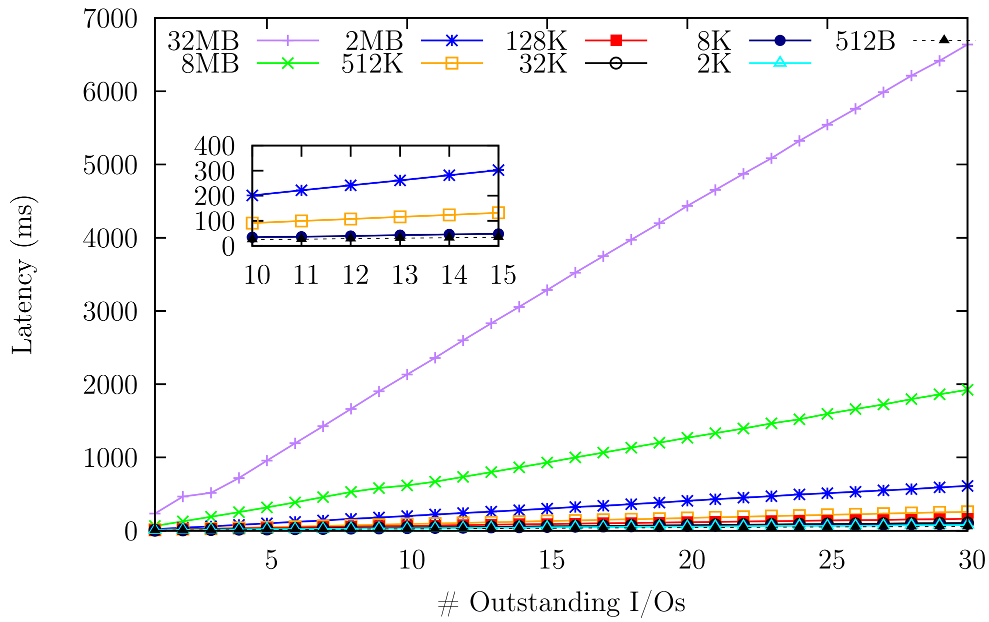


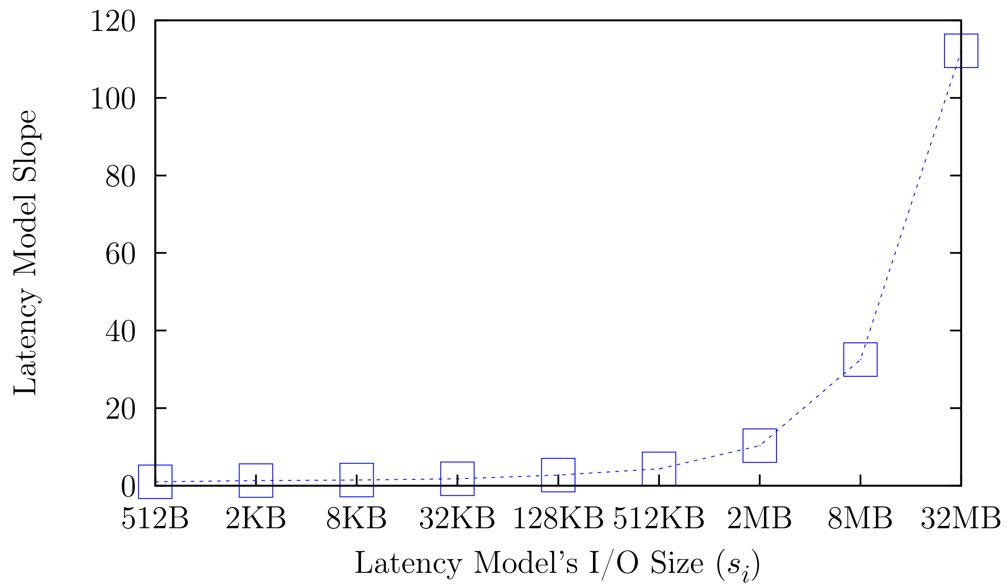
Figure 3.3: Throughput models for a PFS storage node

For example, the throughput model of the data server used for this dissertation's experimental evaluation is shown in Figure 3.3. To confirm that I/Os with different sizes can lead to different values of the $\mathscr{D}$ parameter, the figure shows the throughput model for different I/O sizes, although in practice only the throughput model for small I/Os is required to determine $\mathscr{D}$ for SFQ(D)+. The model is created by issuing random I/O requests directly to the storage, using `DIRECTIO` to bypass the page cache, and using `libaio` to generate different numbers of outstanding I/Os. The results demonstrate that the storage saturates at different points for different I/O sizes. $\mathscr{D}$ is 1 for I/Os larger than 512KB, and between 10 to 15 for I/Os smaller than 512KB.

Second, the *latency model* that captures the relationship between the I/O latency and the number of outstanding I/Os is built to determine the $\mathscr{L}$ parameter of the storage system. In general, a large I/O spends the majority of its time on data transfer which grows with the I/O size, whereas a small I/O spends the majority of its time on request processing which is relatively independent from the I/O size. Therefore, by comparing the latency



(a) Latency models.



(b) The slopes of the latency models in Figure 3.4a.

Figure 3.4: Latency models for a PFS storage node

model of different I/O sizes, it is possible to determine the I/O size that differentiates large I/Os from small ones and determine the slot cost of large I/Os with respect to small ones.

Figure 3.4 shows the latency models for different I/O sizes from the same profiling experiment discussed above. The results confirm that the latency models of small I/Os are similar, whereas the models of large I/Os are quite different. To make this observation clearer, Figure 3.4b plots the slopes of all the latency models in Figure 3.4a. It further confirms that the latency slope does not change much for I/O sizes less than 128KB, but it grows proportionally for larger I/O sizes. These latency profiling results show that 128KB is an appropriate value from differentiating small I/Os from large ones, where all I/Os smaller than 128KB can be considered small I/Os and have the same unit slot cost $\mathscr{L} = 1$. Moreover, the results also help determine the slot cost of large I/Os. For example, if the typical large I/Os that the storage serves is 2MB, then the ratio between the latency of 2MB I/Os vs. the latency of small I/Os decides the slot cost of large I/Os, which is in this case 5. It means that when the scheduler dispatches an I/O smaller than 128KB, it uses only one of the $\mathscr{D}$ slots that the underlying storage has; but to dispatch an I/O of 2MB, it has to use 5 out of the $\mathscr{D}$ slots. Note that although small I/Os may come with different sizes, large I/Os are often of fixed size determined by the basic block size that the parallel file system is configured with. For example, in PVFS2 the parallel I/Os to data servers are often issued in 256KB data chunks.

As discussed above, the two key parameters, $\mathscr{D}$ and $\mathscr{L}$, of SFQ(D)+ can be determined using simple profiling experiments which need to be done only once as the throughput and latency models do not change for a given HPC storage stack. In this way, SFQ(D)+ is able to understand the actual capacity of the underlying storage for processing different I/Os and the actual costs of these I/Os, in order to achieve fair sharing of the storage for I/Os with diverse sizes.

Algorithm 1: Dispatching Algorithm of SFQ(D)+ Scheduler with Backfilling

```
Procedure Dispatch()
    // This function is invoked upon the arrival of a new request
       or the completion of a dispatched request
    Output: The I/O requests to be dispatched IOs_to_dispatch
1   foreach request r in the scheduler queue do
        // The queue is sorted in the ascending order of the
           requests' start times
        // L_sum is the total slot cost of dispatched requests
        // r.L is the slot cost of r
2       if r.L + L_sum <= D then
3           IOs_to_dispatch.add(r)
4           L_sum <- L_sum + r.L
5       end
6   end
7   return IOs_to_dispatch
```

3.4.3 Backfill I/O Dispatching

A side-effect of variable I/O slot costs is that the request with the highest priority in the scheduler's queue may not be able to be dispatched immediately even when there are remaining depth slots available but less than the slot cost of the request. The request can be dispatched only when the other outstanding I/Os complete and the number of available slots is restored beyond its slot cost. When there are unoccupied depth slots, the underlying storage is also underutilized. To make efficient use of the storage resources and increase the overall system throughput, SFQ(D)+ adopts an optimization to dispatch I/Os with backfilling so that small I/Os can enter the storage before the large I/Os queued ahead of them when there are idle depth slots available. The use of backfilling in the algorithm is described in Algorithm 1. This optimization matches the work-conserving nature of the SFQ family of schedulers. At the same time, it further promotes the dispatch of small I/Os and is consistent with the SFQ(D)+ algorithm's principle of protecting small I/Os which are vulnerable to the contention from large I/Os.

Backfilling [JN99] was originally designed to utilize system capacity in a batch job scheduler. When jobs are scheduled based on their order in the queue, the system capacity may not be fully utilized when the remaining capacity is not sufficient to run the job that is currently at the head of the queue. Backfilling allows the scheduler to search the queue for small jobs that fit the remaining capacity and immediately run them in order to fully utilize the system. Akin to the backfilling of jobs, the backfilling for I/O scheduling allows small I/Os to be dispatched before the large ones that are queued ahead of them in order to fully utilize all the slots of the underlying storage's I/O depth. Specifically, whenever there are free slots available but they do not fit the I/O that is currently at the head of the queue, the SFQ(D)+ scheduler searches its queue for a small, fitting I/O to dispatch instead of waiting for enough free slots to be available for dispatching the large request at the head of the queue. Note that when searching for backfill candidates, the algorithm follows the start-time order of the requests in the queue and stops when the available free slots are used up or there is no queued request that fits the remaining slots. Therefore, while promoting small I/Os, the scheduler still maintains fairness among small I/Os from different applications.

Overall the backfilling I/O dispatching technique complements the variable I/O slot cost in SFQ(D)+ and helps applications with small I/Os mitigate the impact from large, intensive applications in two key aspects. First, when the depth of the underlying storage is fully utilized, only the completion of a large outstanding I/O can warrant the entrance of a similar large I/O, therefore avoiding the dispatch of too many large I/Os to impact the small I/Os. Second, when large I/Os cannot be dispatched due to the lack of sufficient slots, small I/Os queued by the scheduler can be dispatched out of order. Therefore, the combination of these techniques in the new SFQ(D)+ scheduler provides the necessary support for achieving fair scheduling in modern, HPC storage systems which face mixed workloads with highly diverse I/O sizes.

3.5 Parallel Storage Distributed Scheduling Coordination

3.5.1 Total-Service Proportional Sharing

The scheduling algorithm described in the previous section supports the proportional sharing of the bandwidth of individual data servers and metadata servers in a parallel file system, and is employed by every vPFS proxy to enforce the bandwidth allocation of its local data and metadata storage. But because an application's parallel I/Os requires service from different servers, it is desirable to provide *total-service fairness*, where the total amount of I/O service that an application gets from all the distributed servers in the system is proportional to its weight. However, local proportional sharing at every parallel file system server is not sufficient to derive proportional sharing of the entire system's total bandwidth, because the I/O service distribution can be uneven across the servers, due to uneven distribution of an application's I/Os and uneven distribution of I/O contention from different applications on each server. As a simple example, consider two applications *A* and *B* sharing the parallel file system, where *A* has its data striped across all the data servers but *B*'s data is only on half of the servers. Even if each server enforces equal sharing of the local bandwidth between *A* and *B*, the total bandwidth that they get from the entire system will not be equal—*A* in fact gets twice the amount of total bandwidth than *B* because *A*'s I/Os are serviced by twice the number of servers.

DSFQ [WM07] is a distributed SFQ scheduler that supports total-service proportional sharing in a distributed storage system by employing a set of coordinators between the storage clients and servers. Each client uniformly accesses all the coordinators, giving each coordinator equal chance of collecting request statistics from all the applications. Each coordinator has two roles, forwarding requests to their destination servers, and keeping track of each application's requests. When forwarding a request to a server, the coordinator also piggybacks the cost of requests from this application forwarded to the other

servers. When the server receives the request, its local scheduler becomes aware of the application's total service received from the entire system, and adjust the local scheduling (by shifting the start time of the request by the piggybacked cost and hence delaying the processing of the request) to achieve total-service fairness.

However, it is difficult to apply the distributed coordinators scheme used in DSFQ to the management of an HPC parallel storage system, because it assumes that 1) each coordinator can forward requests destined to any server, and 2) each application uses all coordinators to forward requests. These assumptions are to ensure that a coordinator can communicate with all local schedulers and always has a uniform chance to piggyback the global scheduling information for each local scheduler, but they do not hold in typical HPC parallel storage systems. For high-throughput, a parallel file system client always issues data requests directly to the data servers where the corresponding data is stored on (after retrieving the data layout from the metadata server). Therefore, there is nowhere in the HPC system architecture to place the required coordinators which can receive requests from arbitrary clients—regardless of the data layouts—and forward them to arbitrary data servers.

It is possible to modify this architecture to make the data layout opaque to the clients, place the coordinators between the clients and servers, and then force the I/Os to go through the coordinators in a random fashion. However, this design would still be undesirable because 1) it requires a coordinator to forward requests to remote data servers and thus incurs overhead from additional network transfer; 2) it takes away an application's flexibility of specifying data layout, e.g., by specifying layout hints through the MPI-IO interface [TGL96]. These constraints imposed by typical parallel file system architecture motivate the need for a new distributed scheduling design that supports both efficient data access and total-service fairness.

To achieve total-service proportional sharing, vPFS takes a distributed scheduling approach where the local SFQ(D)+ schedulers implemented by the proxies on the data and medadata servers communicate with one another to obtain global I/O service information and adjust their local service allocations accordingly to achieve the desired total-service fairness. Compared to the approaches taken by DSFQ, this distributed scheduling design is suited for typical parallel storage architecture, because, first, it still allows data to directly flow from clients to servers, and second, it does not require a proxy to forward requests to any other remote servers than it local one. However, the need of efficient global scheduling synchronization across the distributed proxies remains to be a challenge which is addressed by the techniques presented in the rest of this section.

3.5.2 Threshold-driven Global Scheduling Synchronization

Strictly speaking, global synchronization is required upon every request that a local scheduler receives so that the other schedulers can adjust their scheduling immediately. However, such a per-request broadcast scheme is not acceptable in a large system. Instead, vPFS considers schemes to reduce the frequency of broadcast by batching the costs of a number of locally serviced requests in a single broadcast message. Nonetheless, as a tradeoff, the fairness guarantee offered by the original DSFQ algorithm [WM07] may be weakened. The overhead of broadcast can be effectively controlled if the proxies synchronize with one another periodically. but this scheme does not provide a bound on the amount of unsynchronized I/O cost across proxies. As a result, if a server services a large number of requests during a synchronization period, it would cause high fluctuations on the other servers as they try to catch up with the total-service fairness after the synchronization.

Algorithm 2: Global I/O Cost Synchronization

```
Procedure UpdateAndTrigger(p^i_{f,A})
    // Update the accumulated cost and trigger a synchronization
       message if necessary
    Input: p^i_{f,A}, the i^th request from App f on server A
1   cost_f^sum ← cost_f^sum + p^i_{f,A}.cost
2   if cost_f^sum > threshold_f then
3       Message ← new HashMap()
4       foreach App j do
5           Message.put(j, cost_j^sum)
6           cost_j^sum ← 0
7       end
8       foreach server A serving App f do
9           SendMessage(A, Message)
10      end
11  end

Procedure ReceiveAndSync(Message)
    // Receive the synchronization message and update the local
       scheduling parameters
    Input: Message received from another vPFS scheduler
1   foreach app f in Message.keySet() do
2       batchcost_f ← batchcost_f + Message.get(f)
        // batchcost is the amount of service that App f has received
           from the other schedulers; it is used to delay the start
           time of f's requests in the queue
3   end
```

In order to achieve efficient total-service proportional sharing with a good unfairness bound, vPFS adopts a *threshold-driven global synchronization* scheme. In this new scheme, a broadcast message is triggered whenever the accumulated cost of arrived requests from an application f on the local server exceeds a predetermined threshold. In this way, the degree of divergence from the perfect total-service fairness is bounded by the threshold , because no application would get unfair extra service more than this bound. This scheme can also limit the fluctuation after each synchronization to the extent of the threshold. The pseudo code for threshold-driven global synchronization trigger is illus-

trated in Algorithm 2. A vPFS scheduler uses the `UpdateAndTrigger` procedure to trigger a global synchronization when the accumulated cost from an application exceeds its threshold. A scheduler uses the `ReceiveAndSync` procedure to handle the synchronization message sent from another scheduler. Line 1 accumulates the total I/O service that an application f has received from the other schedulers in $batchcost_f$. When considering the next request from f in the queue, t he scheduler delays its start time by $batchcost_f$, thereby enforcing the total-service fairness betwe en f and the other applications [WM07].

When implementing this threshold-driven scheme, it can be simplified to use a single threshold for all applications on all distributed schedulers, instead of one threshold per application. A scheduler issues synchronization when the total cost of requests from all of its serviced applications exceeds this threshold, where the broadcast contains the costs of all of these applications. This simplification will in fact tighten the unfairness bound. Moreover, this single threshold can be conveniently and flexibly adjusted to balance the tradeoff between efficiency and fairness of the total-service proportional sharing.

3.5.3 Layout-driven Total-Service Proportional Sharing

Although the above threshold-driven total-service proportional sharing can substantially reduce the overhead from global scheduling synchronization, the cost and delay of synchronization will still grow as the number of servers increases in the system. To further reduce the synchronization cost, vPFS also supports a *layout-driven global synchronization* scheme in which each distributed local scheduler leverages an application's file layout information to approximate its global I/O cost from its locally received I/Os. Therefore, frequent global synchronization can be avoided whereas broadcast is required only upon the arrival and departure of applications in the storage system.

A file's layout information, which includes striping method and parameters, can be either discovered in an I/O request or retrieved from a metadata server. For example, PVFS2 embeds the striping method and the specific parameters for this method in every I/O request; if a parallel file system does not do that, such information can still be obtained from the metadata servers. Based on the striping method used by an application, a local scheduler can estimate the application's total service from the striped I/O that it receives locally. For example, in PVFS2, three native striping methods are implemented: simple stripe, two-dimensional stripe, and variable stripe. When using simple stripe, the total service amount can be approximated by multiplying the request size seen by the local server and the number of servers involved in this application. When using two-dimensional stripe, each group's I/O size can be constructed by using the factor number (a number indicating how many times to stripe within each group of servers before switching to another group) within each group to approximate the total I/O service. When approximating the total service in variable stripe, different servers' stripe sizes can be used to reconstruct the original I/O request size.

Another necessary parameter for estimating total service is the number of servers involved in each I/O request. The *num_servers* field is embedded together with the stripe information in the PVFS2 I/O requests. In case it is not available in other parallel file system protocols, only one synchronization is required to obtain this information when a new application enters the system. Although it is possible for an application to use different layouts for its files, it is rarely used in practice. For manageability, the application usually prefers using a uniform layout on its entire data set. Therefore, the layout information needs to be retrieved on a per-application basis rather than per-file basis.

By locally calculating total I/O service using the stripe method and parameters as well as the number of involved servers, the global scheduling synchronization can be almost eliminated. The reduction in broadcast frequency and message size can lead to substantial

saving in processing time and network traffic. Because synchronization is needed only when an application starts or ends in the HPC system, this cost is negligible compared with the typical time that the application stays in the system. Note that the arrival and departure of applications can be estimated by tracking I/O requests or informed by a typical job scheduler (e.g., TORQUE [Sta06], LoadLeveler [SCZL96]) commonly used for HPC job management.

Even for applications using mostly small, non-striped I/Os, it is common for the parallel file system to evenly distribute small I/Os on all the involved servers for the sake of performance. Hence, it is still feasible to use the layout information to estimate the total service in such systems. In scenarios where this assumption does not hold, vPFS can switch to use the threshold-driven synchronization scheme for more accurate scheduling of such I/Os. To enable the switch, each local scheduler monitors the amount of small I/Os that it receives and when the accumulated cost of these I/Os exceeds the predefined threshold, it triggers a synchronization with the other local schedulers as described in Section 3.5.2. Therefore, the above two synchronization schemes complement each other and can be used in different scenarios to provide effective total-service proportional sharing.

3.6 Parallel Storage Metadata Scheduling

he above discussion on the SFQ(D)+ scheduler focused on the scheduling of data I/Os on a parallel file system. As the dataset of modern HPC workloads grows, they are also increasingly metadata-intensive [MBL+11, CLR+09, PG11, AZ11, JSPW+11, MBO+11, BGK+12, AEHH+11, LRT04, LWG+15], which warrants considerations for the performance of metadata I/Os as well. In fact, it has been observed that the amount of metadata in a system grows at a faster rate than the data. Taking WRF, a real-world scientific application as an example, it entails a variety of intensive metadata operations. First, for

checkpointing, WRF uses file-per-process, N-to-N access pattern, in which each process writes to its own checkpointing file. The creation of and accesses to a large number of checkpoint files involve substantial metadata operations, which, if not serviced with the adequate throughput, may slow down the checkpoint operations and delay the application progress. Second, for scalable output operations, WRF often employs a group of dedicated I/O processes (in the *I/O quilting* mode) [WB05] to perform I/Os on a shared output file. These output files involve a large number of concurrent operations to the shared metadata, which can also become a bottleneck if they are not provisioned with the fair share on throughput.

As an example, Figure 3.5 shows the numbers of metadata I/Os versus data I/Os during a typical run of WRF on PVFS2. A total of 64 MPI processes were spawned to execute WRF on 8 nodes, and they used POSIX to access the parallel file system (via the mount points) which includes 8 nodes serving as both data servers and metadata servers. To fulfill this job, the parallel file system used a large number of metadata I/Os in addition to the data I/Os. The number of metadata I/Os actually far exceeds the number of data I/O requests. During the whole run, metadata requests must accompany data I/O requests to fulfill the job. For example, the metadata I/Os include many GETATTR requests issued by the PVFS2 clients for retrieving the attributes of the input, output, and restart files throughout the run. Each request has an average size of 176 bytes. Meanwhile, most of the data I/Os are 512KB or 64KB in size and split into 64KB or 8KB parallel requests, respectively, across the servers. While this example is specific to PVFS2, as discussed above, parallel file systems in general can have intensive metadata I/Os which are critical to application performance and need to be effectively scheduled for performance management.

Although modern parallel file systems have adopted various techniques to improve the scalability of metadata management, they are not sufficient to guarantee throughput to

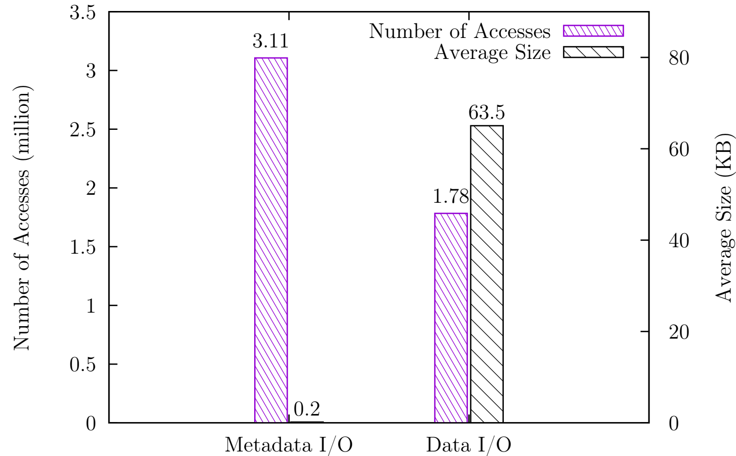


Figure 3.5: Metadata I/Os and data I/Os comparison

metadata-intensive applications. For example, parallel file systems can employ distributed metadata servers to share the metadata load, and there is related work on optimizing the load balance across the distributed metadata servers [PG11]. Some systems start to deploy solid-state storage to hide the latency increase due to the burstiness of metadata accesses [WUA+08]. However, such scalability techniques can only improve the overall metadata access performance for applications, and cannot isolate the performance interference and provide fair sharing of the metadata servers among competing applications. In fact, these parallel file systems do not provide any means to differentiate metadata requests from different applications, an application's metadata access may be slowed down in unpredictable ways when under contention. Moreover, as an application's data I/Os often depend on its metadata I/Os, the lack of performance differentiation on metadata requests can impact data requests and hurt the fairness of sharing the data servers, which further aggravates the application-level unfairness and loss of performance guarantee.

To solve this problem, the new SFQ(D)+ scheduler can be applied to the scheduling of metadata accesses, thereby providing complete support to fair sharing of key parallel storage resources, including both data and metadata servers, and offering comprehensive I/O performance guarantee to HPC applications. Metadata accesses are similar to small

data requests in size. But different from data requests, the exact size of the response to a metadata request is not always known before scheduling (e.g., the total number of objects in a directory is unknown to the scheduler until the directory content is fetched from the server). Nonetheless, the costs of the generally small metadata accesses are similar in the metadata servers, and it is much more important to measure metadata throughput in terms of IOPS than MB/s. For example, considering WRF in the above example, metadata servers receive millions of requests in a short period with less than 200 bytes of response each. Thus, SFQ(D)+ uses a constant value (10KB) to approximate each metadata I/O's *cost*, which is large enough to cover most of the metadata I/Os and small enough with respect to 128KB threshold discussed in Section 3.4.2.

Similar to the asymmetric data distribution discussed in Section 3.5, metadata accesses are often unevenly distributed across the metadata severs (despite the best-effort made by the metadata distribution schemes [WPBM04]). In fact, the distribution of metadata services can be more uneven than that of data services because each metadata access is independent, whereas the striped data accesses from the same request are often evenly distributed across the involved data servers. For example, PVFS2 [CLRT00] stores a directory object (usually at a middle level in a directory tree) and all its underlying metadata objects on a single metadata server. Thus, a frequently accessed directory can make its corresponding metadata server a hotspot and cause uneven distribution of metadata services across the metadata servers.

The threshold-driven synchronization scheme presented in Section 3.5.2 can be adopted to address the asymmetric service distribution for metadata accesses. Unlike data requests, metadata requests are not striped and there is also no layout information that can be exploited by a local scheduler to infer the global service distribution. Therefore, the layout-driven synchronization scheme proposed for data I/O scheduling does not apply to metadata I/O scheduling. Instead, vPFS uses the threshold-driven synchronization

scheme to measure and synchronize the metadata service rates for the contending applications across all the distributed metadata servers. But different from data service, the scheduler uses the number of serviced metadata accesses, rather than number of serviced bytes, to set the threshold for triggering a synchronization. Thus, fair sharing of metadata servers can be provided to applications in terms of IOPS (instead of MB/s) which is what really matters to the performance of metadata-intensive applications.

3.7 Experimental Evaluation

3.7.1 Setup

The vPFS approach was prototyped on PVFS2 [CLRT00] for the experimental evaluation. PVFS2 is a modern parallel file system implementation, and has comparable performance to other commonly used implementations [SMMB08]. It has been often used as the platform for studying various parallel file system problems [CLR+09, CST+11, ZDJ11, ZDJ10, SMMB08, MBO+11, PG11, AZ11, SYS+11, RI01, YLP05, OHS+05, ZDJ11, LRT04, AEHH+11]. The fundamental techniques (virtualization-based parallel I/O scheduling) and algorithms (SFQ(D+)) in vPFS for managing the performance of parallel file system I/Os are generally applicable to different specific parallel file system implementations.

This evaluation was done on a test-bed consisting of two clusters, one as compute nodes running a variety of benchmarks and the other as storage nodes running vPFS proxies and PVFS2 (version 2.8.2) servers. The storage cluster has eight nodes each with two six-core 2.4GHz AMD Opteron CPUs, 32GB of RAM, and two 500GB 7.2K SAS disks. The compute cluster has eight nodes each with two six-core 2.4GHz Intel Xeon CPUs, 24GB of RAM, and two 1TB 7.2K SAS disks. Both clusters are connected to the

same Gigabit Ethernet switch. All the nodes run the Debian 4.3.5-4 Linux with the 2.6.32-5-amd64 kernel and use EXT3 (in the journaling-data mode, unless otherwise noted) as the local file system.

The evaluation considers four types of benchmarks:

1. **Data-intensive parallel I/O benchmark**—*IOR* (2.10.3) [SS07], a typical HPC I/O benchmark, is used to generate parallel I/Os through MPI-IO. IOR can issue large sequential reads or writes to represent the I/Os from accessing checkpointing files, which is a major source of I/O traffic in HPC systems [PD04]. It is extended by the author to issue random reads and writes and represent other HPC I/O patterns. Since there is no computation involved, IOR generates the most intensive checkpointing I/O workloads.

2. **Metadata-intensive parallel I/O benchmarks**—*multi-md-test* from the PVFS2 suite is used to represent applications that are metadata intensive. Multi-md-test measures the performance of concurrent parallel metadata operations from processes that use the POSIX interface to the parallel file system mounted on VFS on the computer nodes. It simulates a burst of back-to-back metadata requests issued to parallel file systems with no computation in between.

3. **Scientific application benchmark**—*BTIO* (Block Tri-diagonal solver with I/O subtypes) benchmark from the NASA Parallel Benchmark (NPB) suite (MPI version 3.3.1) [BBB+91] is used to represent a typical scientific application with interleaved intensive computation and I/O phases. The problem solving algorithm and its implementation in BTIO highlight the limitations to I/O performance of the system, and the benchmark is a good case for testing the speed of parallel I/Os [WdW03]. This dissertation considers the diverse I/O access patterns (*Class A* and *Class C*) of BTIO. *Class A* generates 400MB of data and *Class C* generates 6817MB. *Class A*

uses *simple* subtype and *Class C* uses *full* subtype of BTIO. The former does not use collective buffering and as a result involves a large number of small I/Os. The latter uses MPI I/O with collective buffering which aggregates and rearranges data on a subset of the participating processes before writing it out.

4. **Real-world scientific application—***WRF* [WB05] version 3.3, a weather forecast model application widely used for weather research. WRF uses MPI to coordinate parallel computing while using POSIX I/Os for data accesses to the parallel file system mounted on the compute nodes. It is compiled using the *dmpar* configuration (distributed memory option (MPI)), and run with the *em_quarter_ss* test case. This test case produces a simulation of a supercell thunderstorm. The environmental wind makes a "quarter circle" when plotted on a hodograph, and is commonly referred to as "quarter circle shear". In general, WRF simulates and forecasts weather conditions based on models. All WRF test cases share the same I/O pattern: WRF sequentially reads input, and sequentially writes output and checkpointing data periodically during the mass calculation for each granular time unit calculated and forecasted. Although the evaluation here focuses on one specific case, it is representative of the common I/O patterns of all WRF cases and the results are thus useful to WRF users in general. In this setup, WRF starts with about 50MB of input data, and for every forecast minute during the storm's progress, writes to an output file shared by all processes and a restart file for each process. The entire execution generates a total of 70GB of restart files (from 64 processes) and 7GB of final output file. Input, output, and restart files are all stored using NetCDF3 [RD90], a commonly used scientific data format. Average I/O sizes, after striping is around 32KB on each data server. Note that the choice of data format for WRF will not change the relative comparison between vPFS and the native parallel file system,

because the management of data and metadata I/O contention is already needed regardless of the data format used.

The rest of the chapter includes several groups of experiments designed to evaluate various key aspects of vPFS: Section 3.7.2 evaluates its ability to provide fair sharing among highly I/O-intensive workloads with different access patterns; Section 3.7.3 studies how it handles dynamic workloads and resource availability in the system; Sections 3.7.4 and 3.7.5 evaluate the support for scientific applications with both large and small I/Os; Section 3.7.6 focuses on the scheduling of metadata-intensive applications; finally, Section 3.7.7 measures the overhead of vPFS and Section 3.7.8 shows its development cost. Based on the profiling results discussed in Section 3.4.2, the parameter $\mathscr{D}$ (the total number of slots that the storage supports) for SFQ(D) and SFQ(D)+ is set to 14, and the parameter $\mathscr{L}_{large}$ (a large I/O's *slot cost*) for SFQ(D)+ is set to 5, unless otherwise noted.

3.7.2 IOR with Various Access Patterns

This experiment evaluates the ability of vPFS to enforce fair bandwidth sharing between I/O-intensive applications with diverse I/O patterns. IOR is used to represent two applications, *App1* and *App2*, each running 128 processes on a separate set of compute nodes. *App1* always issues sequential writes, whereas *App2*'s I/O pattern changes from sequential writes, sequential reads, to random reads and writes (both the offset of the requests and the use of read versus write are randomly decided following a uniform distribution). To evaluate server synchronization effectiveness, the two applications share the data servers in an asymmetric way: *App1* uses only four of the eight data servers whereas *App2* uses all of them. Without any bandwidth management in such an asymmetric setup, the total bandwidth that each application gets is proportional to the number of servers that it

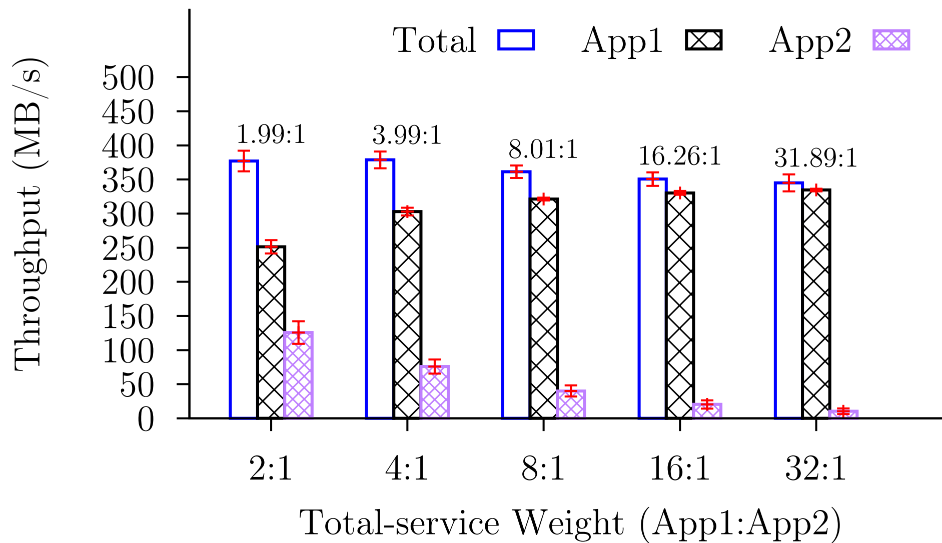


Figure 3.6: IOR write vs. IOR write

has (1:2). Therefore, this experiment can evaluate whether the distributed schedulers can realize total-service fairness for any arbitrary weights assigned to *App1* and *App2*.

Figures 3.6, 3.7 and 3.8 show the throughputs of the two applications with different access pattern combinations and with different total-service weights assignment. The achieved bandwidth share ratios are shown on top of the bars in the figures, which all closely match the given weight ratios across the diverse experiment configurations. These results are obtained from the threshold-driven synchronization method in vPFS, where the threshold is set to 10MB. It achieves the same level of fairness as a much smaller threshold (512KB) which triggers synchronization upon every request (results not shown here for the sake of brevity), while reducing the synchronization overhead by 95%.

Note that the total throughput decreases as the gap between the App1:App2 weight ratio increases. It is due to the asymmetric distribution of the applications' data. When App1 is favored more by the scheduler, App2's I/Os are suppressed more on the four servers that App1 has access to, which in turn also suppresses the amount of service that App2 gets from the other four servers, because an App2's request is always striped across

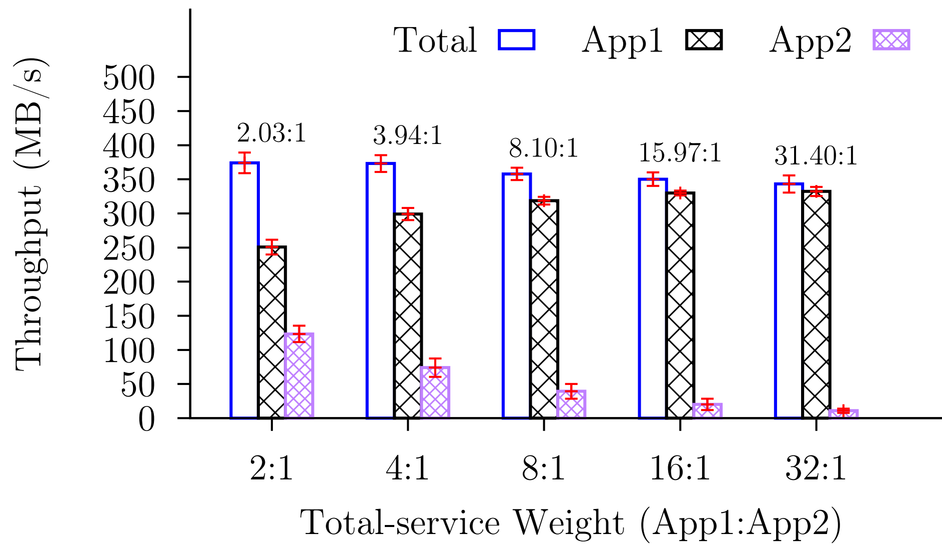


Figure 3.7: IOR write vs. IOR read

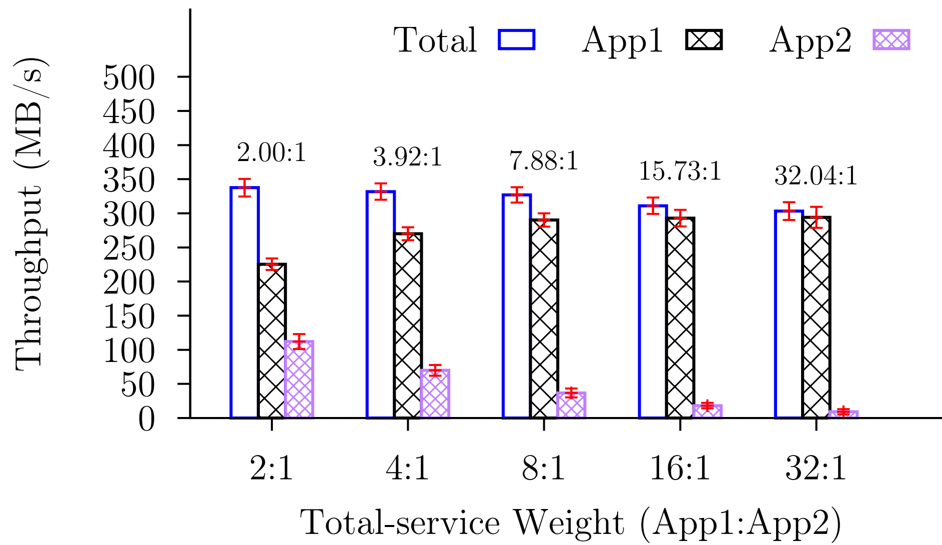


Figure 3.8: IOR write vs. IOR random read/write

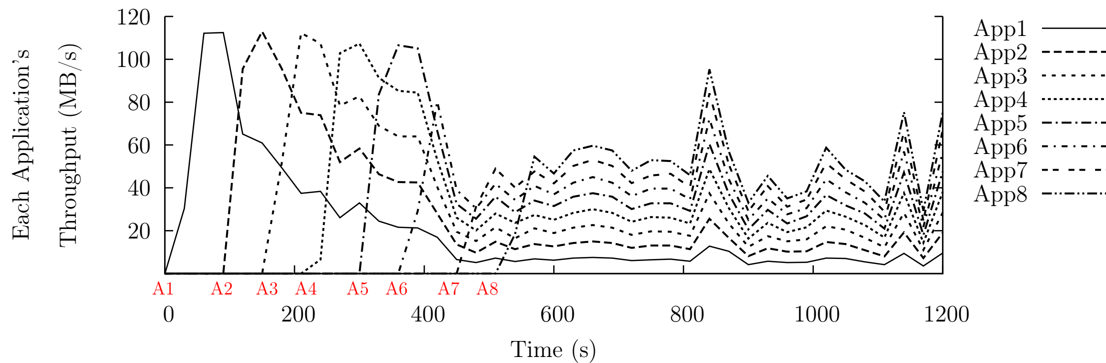


Figure 3.9: The throughput of 8 competing IORs

all eight servers. Therefore, this drop in total system utilization is only a consequence of the experiment setup, not the shortcoming of the scheduling algorithm.dd

3.7.3 IOR with Dynamic Arrivals

This experiment evaluates vPFS' ability of handling the dynamic arrivals and departures of applications which are common in a production HPC system. A total of eight applications enters the system one by one with an average inter-arrival time of about 60 seconds, where each application is represented by an IOR instance with 32 processes in sequential writing mode. After 480 seconds, all eight applications (256 processes) are present in the system until the end (1200th second). To make the experiment more interesting, the odd-numbered applications use only four data servers and the even-numbered ones use all of the eight servers; and the weight of each application is assigned the same value as its ID, i.e., *App1* has a weight of 1 and *App2* has a weight of 2, and so on. Different from the pervious experiment, this experiment employs the layout-driven global synchronization.

Figure 3.9 illustrates the average throughput achieved by each application measured every 5 seconds throughout the experiment. The results show that each application's throughput is indeed proportional to its assigned weight throughout the experiment, in spite of the asymmetric data distribution and the changing number of concurrent applica-

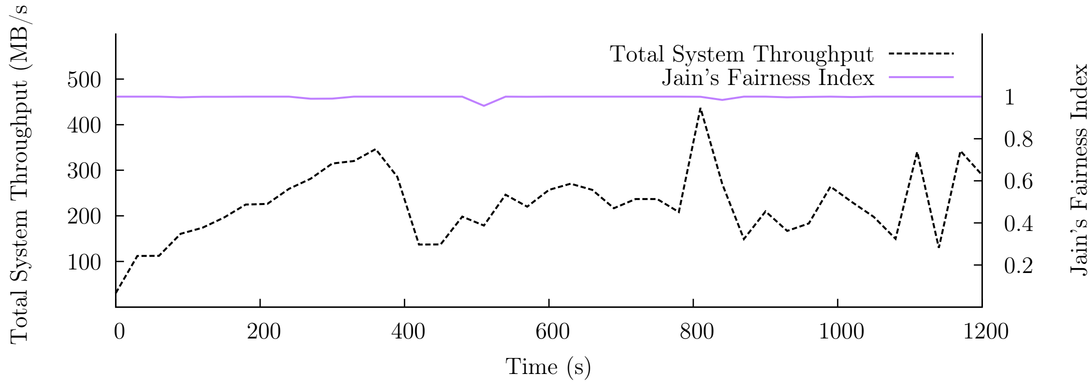


Figure 3.10: The fairness of 8 competing IORs

tions in the system. Figure 3.10 shows the the total I/O throughput and the Jain's fairness index from the eight applications. Jain's fairness index [JCH84] is a commonly used metric for evaluating the fairness in resource sharing. It is defined as $\frac{\left(\sum_{i=1}^{n}\frac{T_i}{W_i}\right)^2}{n\sum_{i=1}^{n}\left(\frac{T_i}{W_i}\right)^2}$ where T_i and W_i are the throughput and weight, respectively, of Application i in the system. The range of the fairness index is $[0,1]$ where a larger index value indicates higher fairness. The results in Figure 3.9 show that vPFS achieves nearly perfect fairness (above 0.99) throughout the experiment, despite the fluctuations in total system throughput as applications enters the system dynamically.

Moreover, these results also show that vPFS is able to handle the intrinsic dynamics of the underlying storage that it manages. At the beginning of the experiment, the total system throughput increases almost linearly as the applications enter the system one by one and start the writes which are initially buffered in memories of the data servers and flushed to disks in background. But after about 500s into the experiment, the total throughput drops significantly, because the memory buffers are filled up and the servers have to flush the writes to disks in foreground. This transition causes a substantial change in the system's total bandwidth, which is bounded by the speed of the memories before the transition and bounded by the disks afterwards. Nonetheless, vPFS is always able to

deliver the specified total-service sharing ratios regardless of such dynamic changes in the underlying storage.

3.7.4 BTIO vs. IOR

This experiment applies vPFS-based storage management to solve the motivating problem described in Section 3.2, i.e., how to guarantee BTIO's I/O bandwidth and overall performance when it is under the intensive interference from IOR. This experiment uses the same setup as the one in Section 3.2, Figure 3.1a, where BTIO is used to model typical HPC applications with interleaved computation and I/O and IOR is used to create intensive contention (e.g., from checkpointing I/Os) on the shared parallel file system storage. BTIO and IOR each has 64 MPI processes running on a separate set of four compute nodes while sharing the eight I/O nodes.

Two types of BTIO workloads are considered in this experiment: *Class C* (writing and reading 6817MB of data) with *full* subtype (using collective buffering) and *Class A* (writing and reading 400MB of data) with *simple* subtype (without collective buffering). A major difference between these workloads is that the former issues I/O requests of 4MB to 16MB in size, whereas the latter issues I/Os of 320B in size. Because of the different levels of small I/Os in the two workloads, this experiment evaluates how effectively vPFS can protect the bandwidth of small I/Os from BTIO against the intensive, large I/Os (32MB writes) from IOR. Note that different from the goal of fair bandwidth sharing in the previous experiments, the goal of this experiment is to provide performance isolation to BTIO while allowing the competing IOR to use the remaining bandwidth in a work-conserving manner, which is challenging to achieve. It also serves a good showcase of vPFS' flexibility of realizing different performance policies.

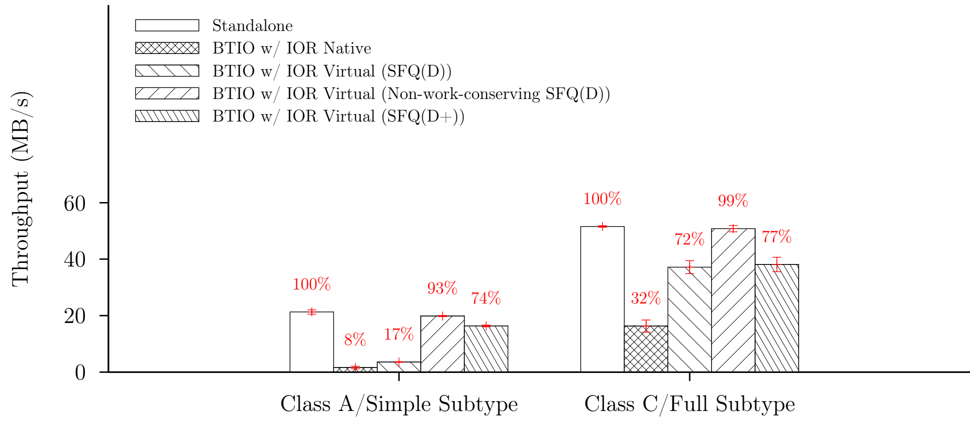


Figure 3.11: BTIO performance restoration

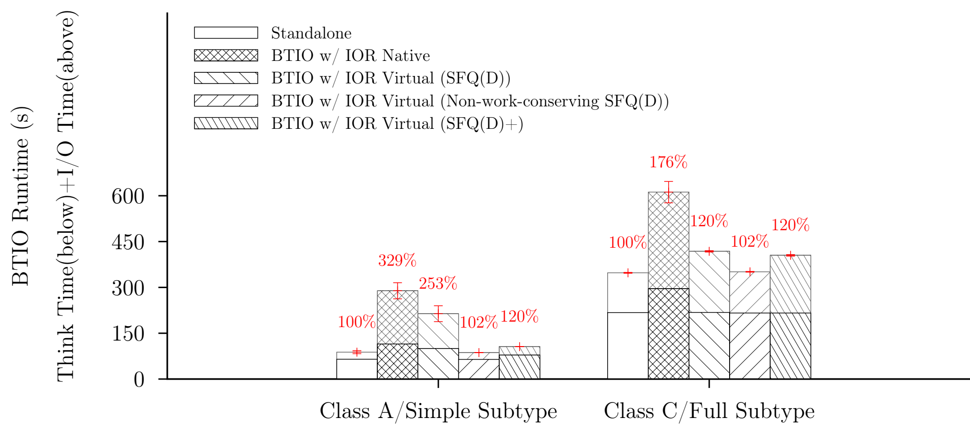


Figure 3.12: BTIO runtime restoration

Figures 3.11 and 3.12 show the I/O throughput and total runtime, respectively, of BTIO under different configurations. As discussed in Section 3.2, when there is no bandwidth management (*BTIO w/ IOR, Native*), BTIO's run time is increased by 228.8% in Class A and 75% in Class C w.r.t. its standalone runtime (*Standalone*). When vPFS employs the traditional SFQ(D) scheduler (*BTIO w/ IOR, Virtual (SFQ(D))*) helps reduce the slowdown to 153.3% for class A and 20.3% for Class C, as the BTIO throughput is restored to 16.8% and 72.1%, respectively, of the *Standalone* case. As discussed in Section 3.4.2, the lack of differentiation between large and small I/Os in the SFQ(D) when considering the dispatch of $\mathscr{D}$ requests puts BTIO at an unfair disadvantage. It is also noticeable that Class A's performance is much more challenging to restore than Class C, because of its use of much smaller I/Os which are also much more sensitive to the interference from IOR's large I/Os.

In order to completely isolate the impact of I/O contention, a non-work-conserving SFQ(D) scheduler is implemented on vPFS, which strictly throttles an application's bandwidth usage based on its allocation. Specifically, this non-work-conserving scheduler will put an application's I/Os temporarily on hold when its completed I/O service exceeds its given bandwidth cap. When running under the non-work-conserving scheduler (*BTIO w/ IOR, Non-work-conserving SFQ(D)*), BTIO can achieve the same level of performance as when it runs alone. Although the non-work-conserving scheduler can provide performance isolation to BTIO, it is not desirable for a shared system because IOR cannot make use of BTIO's spare bandwidth to make progress and the system can be severely underutilized when BTIO's demand is low.

In comparison, the new SFQ(D)+ is designed to protect small I/Os while still being work-conserving. The result (*BTIO w/ IOR, SFQ(D)+*) in Figure 3.12 shows that for Class A, SFQ(D)+ can restore BTIO's throughput to 120% of its standalone case, which is 219% better than SFQ(D) scheduler, and only 18% worse than the non-work-conserving

scheduler. For Class C, SFQ(D)+ cannot achieve more improvement than SFQ(D) because Class C issues I/Os of the same average size as IOR and SFQ(D)+ thus reduces to SFQ(D).

SFQ(D)+ cannot completely restores BTIO's performance as the non-work-conserving scheduler does, because of BTIO's bursty I/Os with low issue rate. Considering one of the 64 BTIO processes, after the initial file creation, the process interleaves 4 seconds of writes and 6 seconds of computation for 40 iterations in the first output phase. Then in the verification phase, it repeats 3 seconds of reads and 1 second of verification for 40 iterations. In addition, each BTIO process issues only one outstanding I/O. Therefore, BTIO's IO issue rate is much lower compared to IOR which issues IOs continuously. Since SFQ(D)+ is work-conserving, spare bandwidth from BTIO has to be yielded to IOR. But when a BTIO process' I/O arrives, it has to wait for a total of $\mathscr{D}$ outstanding I/Os of IOR to complete before it can be dispatched. In comparison, the non-work-conserving scheduler would not dispatch any I/O from IOR until BTIO uses up its fair share of the bandwidth.

However, SFQ(D)+ achieves better balance between resource utilization and performance isolation. When it restores BTIO Class A's performance to 120% of its standalone case, it slows down IOR by only 56% slowdown (from 597MB/s to 262MB/s), whereas the non-work-conserving scheduler has to slowdown IOR by up to 99% (from 597MB/s to 0.69MB/s) in order to provide better isolation for BTIO. Therefore, SFQ(D)+ achieves 13.81 times better total throughput and resource utilization. Moreover, compared to the traditional SFQ(D), SFQ(D)+ achieves 3.35 times better performance for BTIO with only 10.6% reduction on total throughput. Hence, SFQ(D)+ provides excellent performance isolation for a small I/O application while still making efficient use of the shared storage bandwidth.

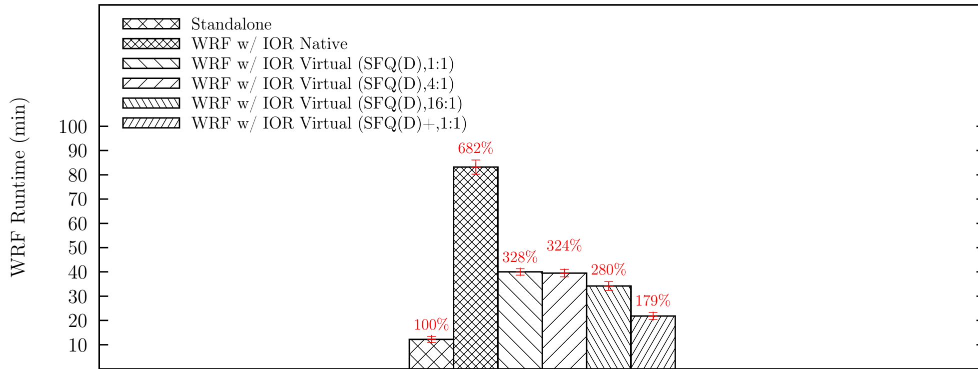


Figure 3.13: WRF runtime restoration

3.7.5 WRF vs. IOR

This experiment continues to evaluate vPFS and its schedulers using WRF, a real-world scientific application. WRF reads input at the beginning of each computation iteration and writes checkpointing data (containing dumps of all varialbes in the model at a certain forecast time) and output data (containing the final forecast variables of the model simulation) periodically. In the experiment, it is run using 64 parallel processes, each issuing reads and writes of sizes 128KB to 512KB in size. IOR is used again to create contention using 64 parallel MPI processes each issuing large (8MB) writes. The two applications are run on separate compute nodes but share all the eight data nodes. Compared with BTIO, it is even more challenging to provide performance isolation to WRF, because it accesses the parallel file system through the POSIX interface which generates all small I/Os and a large volume of intensive metadata accesses (as shown in Figure 3.5).

As shown in Figure 3.13, when contended by IOR, the runtime of WRF is increased by 580+% (*WRF w/ IOR Native*). With vPFS, the SFQ(D) scheduler can reduce the runtime increase to 180% at best when using a high sharing ratio of 16:1 to favor WRF, and SFQ(D)+ can further reduce the slowdown to 79% of the standalone runtime while using a fair 1:1 sharing ratio. This improvement is from the use of variable slot cost to capture

large I/Os' actual processing cost in the underlying storage and the use of backfilling to allow small I/Os to be promoted. For WRF, these techniques benefit both its data and metadata I/Os when they are scheduled on the eight PVFS2 nodes which serve as both data and metadata servers. Overall SFQ(D)+ allows WRF to run up to 81% and 281% faster than SFQ(D) and the native case, respectively.

This experiment is also used to evaluate the impact of the $\mathscr{L}$ parameter—the slot cost of large I/Os—in the proposed SFQ(D)+ scheduler. Figure 3.14 shows the throughput of IOR and WRF as well as their combined throughput with different $\mathscr{L}$ settings. When $\mathscr{L}$ is too small, the scheduler underestimates the cost of large I/Os, and it reduces to the traditional SFQ(D) which considers large I/Os as expensive as small I/Os when making dispatching decisions. The throughput of WRF is thus is substantially reduced, whereas IOR does not gain much because it can already saturate the storage. When $\mathscr{L}$ is set too large, the scheduler overestimates the cost large I/Os, and behaves more like the non-work-conserving scheduler (discussed in Section 3.7.4) where the share allocated to WRF can be left idle when it is not fully utilized. Consequently, the throughput of IOR drops substantially, but WRF cannot gain much due to its low I/O rate; and the combined throughput drops severely too. The experiment shows that adjusting the $\mathscr{L}$ value can balance between performance isolation and resource utilization. It also confirms that the choice of $\mathscr{L} = 5$ based on the profiling results is indeed optimal as it provides the best tradeoff between fairly treating small I/Os vs. large I/Os and fully utilizing the storage bandwidth. Specifically, SFQ(D)+ performs at least 281% better than the non-work-conserving scheduler and 25% better than the traditional SFQ(D) in terms of total system throughput.

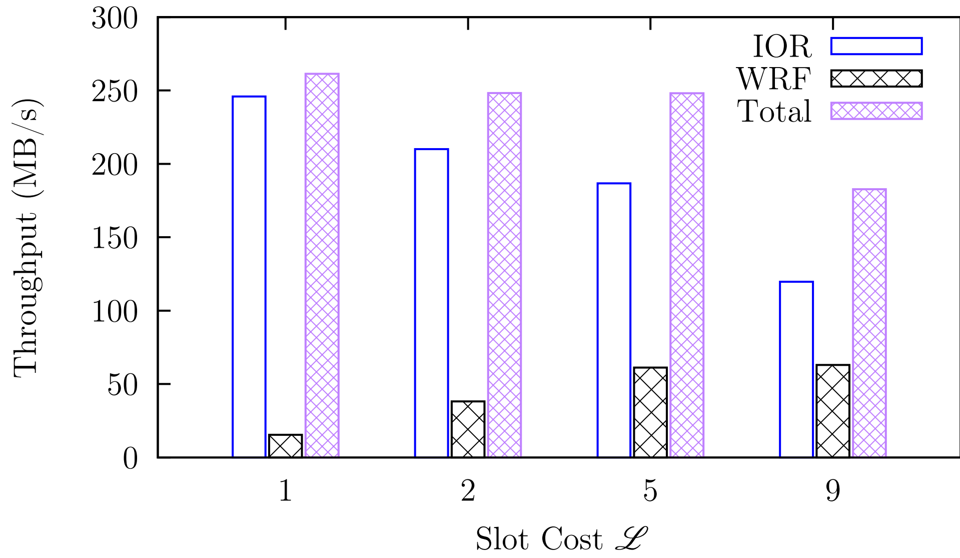


Figure 3.14: Combined throughput of WRF and IOR

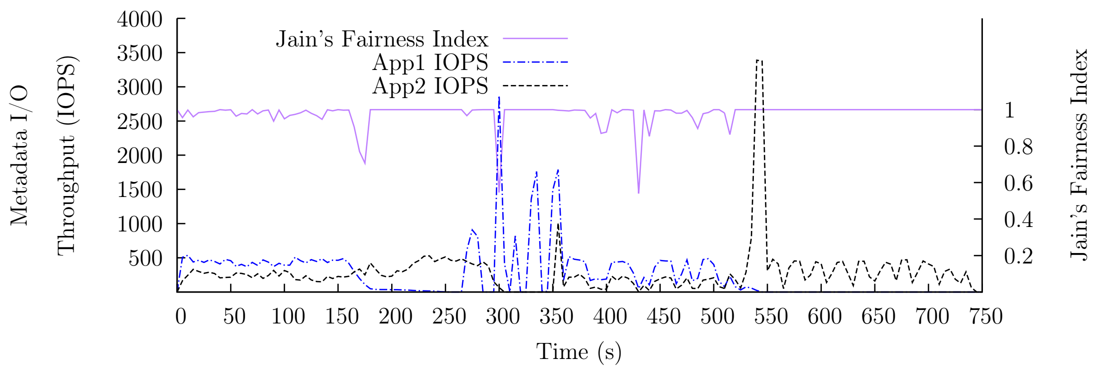


Figure 3.15: Metadata I/O throughput

3.7.6 Metadata Scheduling

This group of experiments focuses on evaluating vPFS' ability of scheduling metadata I/Os and proportionally sharing the bandwidth of parallel file system's metadata servers. Multi-md-test from the PVFS2 suite is used to represent a metadata-intensive application. The PVFS2 setup is configured with distributed metadata servers, where each storage node runs both the data server and metadata server. Two instances of the multi-md-test benchmarks are used, each with 64 parallel processes, to share the eight data/metadata servers. The assigned weights for App1 and App2 are 2:1. Multi-md-test accesses the mounted parallel file system using the POSIX interface, and the PVFS2 clients on the compute nodes convert the POSIX requests to the PVFS2 requests and send them to the distributed servers. Its execution follows the phases of `mktestdir`, `create`, `write`, `readdir`, `read`, `close`, `rm`, and `rmtestdir`. The `write` and `read` phases involve writing and reading of 400MB data to a file per process.

Figure 3.15 shows the metadata I/O throughput of the two applications on one metadata server on the left y-axis and Jain's Fairness Index over time on the right y-axis. (The results on the other metadata servers are similar and omitted here.) The throughput data show two sets of spikes, which represent the high IOPS of the `readdir` phase for both applications, where App1's progress is roughly twice as fast as App2. During the first 160 seconds of the experiment, both applications are performing `mktestdir` and `create` operations, but App1 gets twice the amount of metadata IOPS as App2, which matches their given weights of 2:1. The second overlapped period is from 350 second to 550 second, during which App1 is performing `rm` and `rmtestdir` while App2 is performing `readdir` and `read`. Note that the `read` phase also involves metadata accesses, because a `getattr` request is often issued before a `read` according to the PVFS2 protocol. During this overlapped period, the metadata accesses from the two applications also follow the given 2:1 ratio.

Overall, the Jain's Fairness Index throughout the experiment is above 0.95 in average. There are two periods when one of the two applications is in the `write` phase (160s-260s for App1, 300s-360s for App2) and does not issue any metadata request. During these periods, the Fairness Index drops to close to 0.5 only because vPFS' work-conserving SFQ(D)+ scheduler allows the application that is more active in metadata I/Os to take away the spare bandwidth from the other and make efficient use of the metadata servers' processing capacity.

3.7.7 Overhead

Proxy-based Virtualization and Scheduling Overhead

The last group of experiments studies the performance overhead of vPFS' overhead from several aspects. It first considers the overhead from the proxy-based parallel file system virtualization and scheduling for highly intensive workloads. It compares the throughput between *Native* (native PVFS2 without using proxy), *Virtual* (PVFS2 virtualized using vPFS but without any scheduler), and *Virtual-SFQ(D)+* (vPFS with SFQ(D)+ based I/O scheduling). IOR is used in this experiment to generate intensive sequential reads and writes. The number of processes used by IOR is 256, evenly distributed on eight physical nodes. To increase the intensity of the request rates and demonstrate the worst-case overhead of vPFS, in this experiment only, the file is preloaded into the data servers' memories when IOR runs in the read mode (to eliminate the impact of disk latency), and the file systems on the data servers use EXT3 in the ordered-data mode (to eliminate the impact of data journaling latency). The results in Figure 3.16 show that the throughput overhead caused by the proxy and its scheduler are small, and when they are combined the total is still less than 1% for READ and 3% for WRITE of the native PVFS2 throughput. The extra overhead for WRITE comes from additional message forwarding from the server to

the client to notify the completion of each write request (note that there is a gap between the completion of data transfer from client to server and the completion of writing data on the server side. READ does not bear this overhead because the client automatically completes the request when expected amount of data is received.

The next experiment studies the overhead of metadata I/O virtualization and scheduling in vPFS. A set of eight nodes is used to run a total of 128 multi-md-test processes, each issuing 60K metadata operations, and another set of eight nodes is employed as the PVFS2 metadata servers. Figure 3.17 shows the benchmark's total throughput of metadata operations from vPFS (*Virtual*) versus the native PVFS2 (*Native*). Two different configurations of the benchmark are considered: one issues metadata operations through the POSIX interface (*POSIX*) and the other through the PVFS2 interface (*PVFS2*) directly. The former configuration has lower throughput because metadata operations have to go through additional layers (Linux virtual file system) to reach the parallel file system. Similarly to the previous experiment, the memory caches are kept warm in order to maximize the metadata I/O rate and reveal the worst-case overhead of vPFS. Overall the overhead is less than 3% when using the PVFS2 API and less than 5% when using POSIX. Note that the PVFS2 API does not provide the `close` call, and the POSIX API does not offer `readdir_stat` and `readdir_plus`, which are omitted in the figure.

The resource usage overhead of vPFS is also measured for an experiment with eight concurrent IOR instances competing for eight shared data servers by reading the CPU and memory consumptions of a vPFS proxy running the SFQ(D)+ scheduler with the threshold-driven synchronization scheme (which has higher overhead than the layout-driven scheme). Figure 3.18 compares the proxy's resource usages to the native PVFS2 daemon's, and the results show that the overhead is low and comparable to the native daemon even when it has to handle 256 concurrent IOR processes' I/Os on each data server.

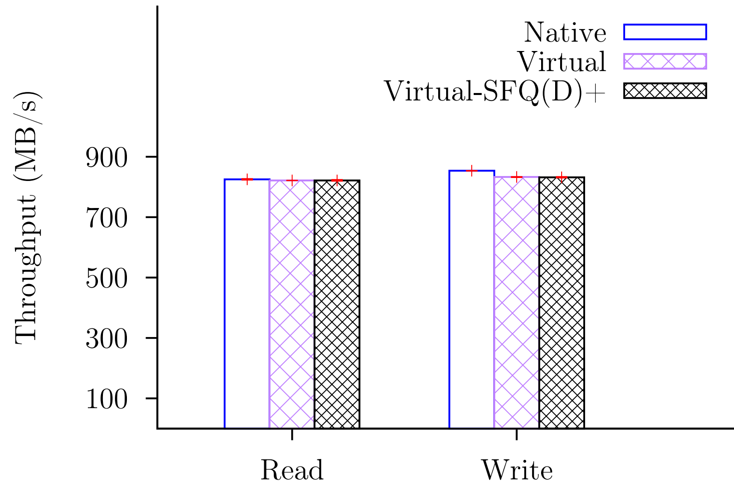


Figure 3.16: The overhead of vPFS for a data-intensive application

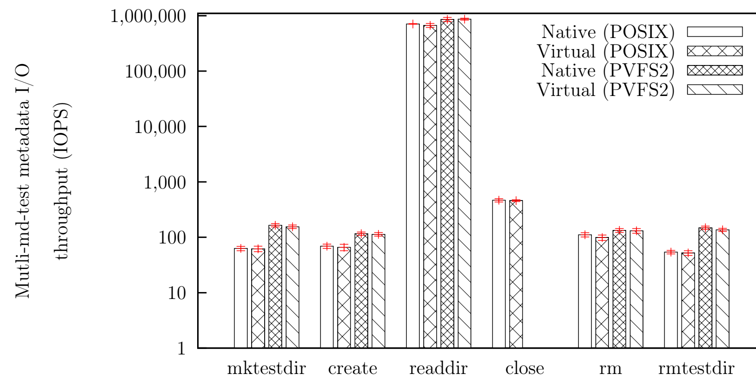


Figure 3.17: The overhead of vPFS for a metadata-intensive application

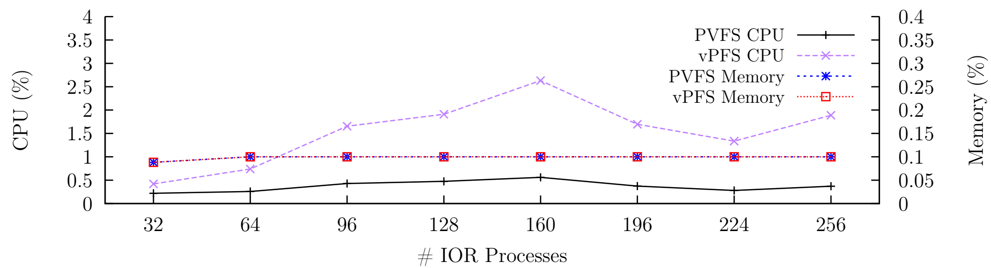


Figure 3.18: CPU and memory overhead of vPFS

The results show that the overhead of vPFS proxy-based virtualization and scheduling is small, which can be attributed to several performance optimizations taken by the proxy implementation. First, the proxy employes single-threaded, polling-based synchronous I/O multiplexing instead of using a multi-threaded model in order to save the cost of context switching. On each proxy (one per server), exactly one polling function is called on an array of sockets. Second, the polling function activates only when there is available data (for read) or available buffer (for write) on the sockets. Third, the proxy processes the data on multiple ready sockets returned by the polling call in batch. It reads/writes each socket in a non-blocking manner, processing only the data that is actually available and returning immediately when the work is done. Fourth, to reduce the memory overhead, the proxy does not store a request's payload when queueing the I/O requests; instead, it extracts the request headers and queues only the information necessary for determining the scheduling action (e.g., IP address, payload size, total number of servers involved, striping parameters, etc.).

Synchronization Overhead

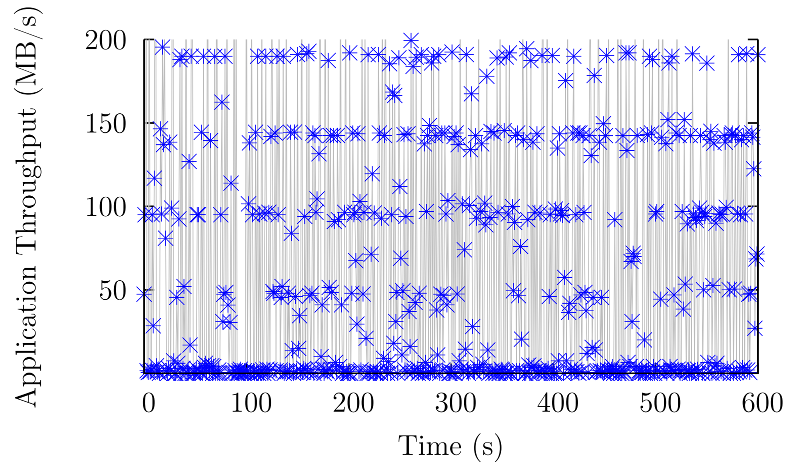
The results from Sections 3.7.2 and 3.7.3 confirm that both the threshold-driven and layout-driven synchronization schemes can achieve equally good sharing of the system's total bandwidth (within 3% of total-service fairness). However, as discussed in Section 3.5, the synchronization overhead differs between these two schemes, which is evaluated here. In order to make this difference more evident, this experiment 1) involves a total of 96 data servers running on Xen virtual machines (with paravirtualized kernel 2.6.32.5) hosted on the eight server nodes, 2) uses the NULL-AIO [Din09] in PVFS2 to skip the actual disk reads and writes and maximize the I/O rate, 3) runs eight concurrent IOR instances, each with 32 parallel processes hosted on a separate physical compute node, and 4) sets the synchronization threshold below the average request size so that every request

triggers a broadcast in the threshold-driven scheme. Note that this setup is designed to reveal the worst-case overhead of the threshold-driven synchronization scheme. In practice, the threshold is set much higher and the corresponding synchronization overhead is much lower.

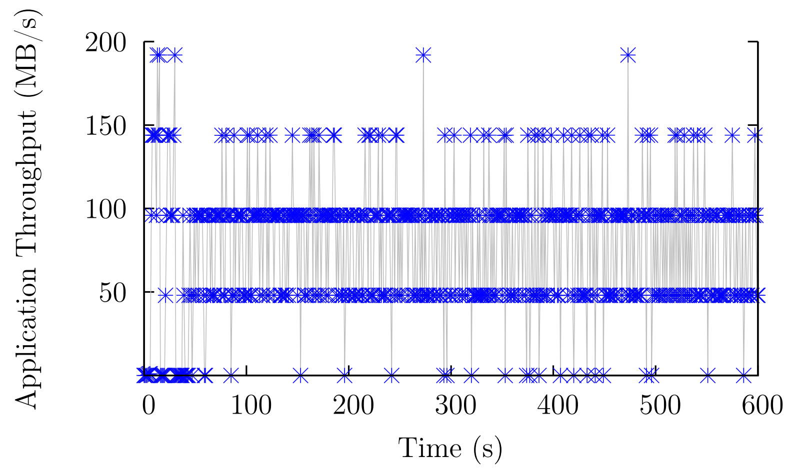
Figure 3.19 shows the throughput of one of the applications and Figure 3.20 shows the synchronization traffic on one of the servers. (The results obtained from the other applications and servers are similar.) In this worst-case setup, the threshold-driven scheme has a constant need of synchronization among the distributed SFQ(D)+ schedulers. Such frequent synchronization also causes high fluctuations on the throughput because every synchronization triggers a local scheduler to adjust the start times of its queued requests. In contrast, the layout-driven scheme involves synchronization only at the entry and exit times of an application. As a result, layout-driven synchronization scheme shows a much smoother total system throughput in Figure 3.19 and much less synchronization traffic in Figure 3.20. As shown in Figure 3.21, the layout-driven scheme also achieves 13.3% higher total throughput (612.08 MB/s vs. 540.91 MB/s) and 93.0% lower deviation (2.21 MB/s vs. 31.59 MB/s) than the threshold-driven scheme. Meanwhile, it can still correctly estimate the total service based on the layout information and achieve good fair sharing (with a Jain's fairness index value of 0.97).

3.7.8 Cost of Implementation

Finally, vPFS is designed as a framework that allows various I/O schedulers to be developed in a modularized fashion and flexibly plugged in for parallel file systems. To evaluate the development effort, Table 3.1 summarizes the code complexity of the vPFS framework. The total lines of C code currently in the prototype sums up to $5,197$, including the support for TCP interconnect, PVFS2 parallel file system, and three types of

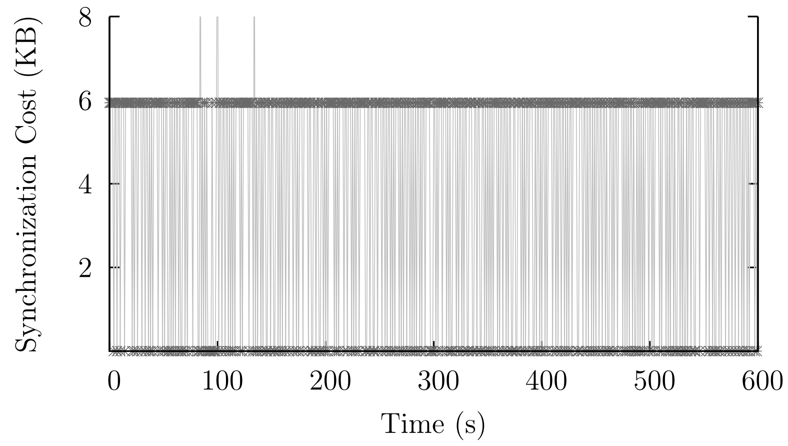


(a) Threshold-driven synchronization scheme

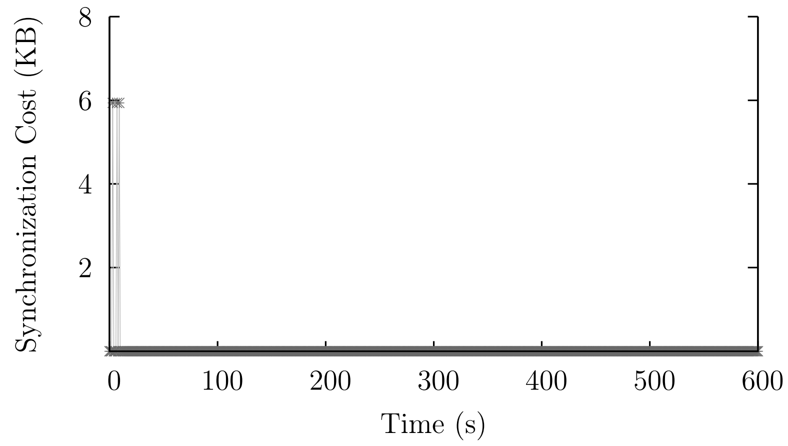


(b) Layout-driven synchronization scheme

Figure 3.19: I/O throughput of different synchronization schemes



(a) Threshold-driven synchronization scheme



(b) Layout-driven synchronization scheme

Figure 3.20: Network traffic of different synchronization schemes

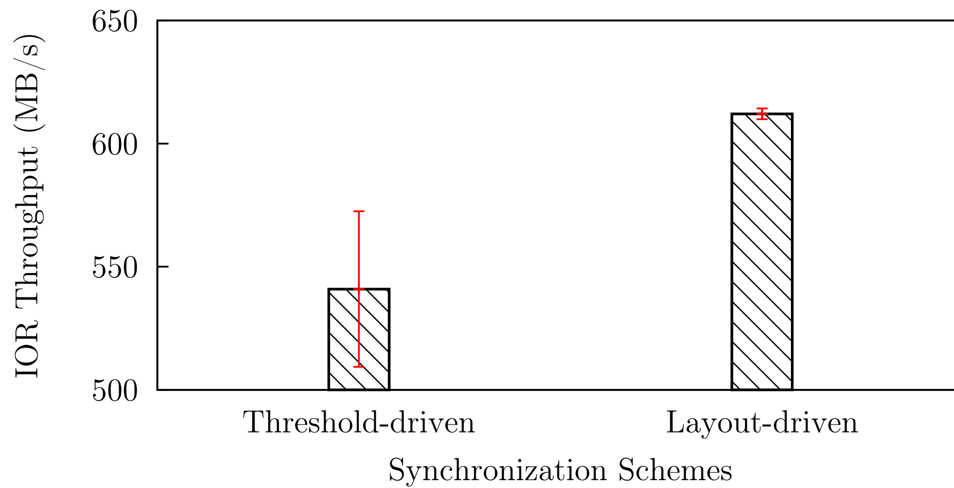


Figure 3.21: Application throughput of different synchronization schemes

Table 3.1: The development cost of vPFS virtualization and schedulers. The lines of code are counted for different components of vPFS.

Framework	**LOC**	**Components**	**LOC**
Virtualization	1695	Interface	694
		TCP	397
		PVFS2	601
Scheduler	3502	Interface	735
		SFQ(D)	552
		SFQ(D)+	987
		Two-Level	1228
Total	5197		

schedulers. To break it down, the virtualization framework costs 1,695 lines of code and the scheduling framework costs 3,502 lines of code. The generic interfaces exposed by these frameworks allow different network transports, parallel file systems, and scheduling algorithms to be incorporated into vPFS. Specifically, the support for TCP and PVFS2 protocols each costs less than 1,000 lines of code. Different schedulers cost from 500 to 1,300 lines of code depending on their complexity. For example, the most complex one is an experimental two-level scheduler [XZ13] that supports both throughput and latency driven I/O scheduling, and it costs 1,228 lines of code.

3.8 Summary

A new approach, vPFS, to parallel storage management in HPC systems is presented. vPFS addresses this problem through the virtualization of contemporary parallel file systems. Upon the vPFS framework, a new proportional sharing I/O scheduler, SFQ(D)+, is proposed to allow applications with diverse I/O sizes and issue rates to share the storage with good application-level fairness and system-level utilization. vPFS also includes a combination of efficient synchronization schemes for a large number of distributed schedulers to coordinate their local I/O scheduling and achieve the global, total-service sharing target.

vPFS approach is feasible because of its small overhead in terms of throughput and resource usage ($<$3% for reads, $<$3% for writes, and $<$3% for metadata accesses). Meanwhile, it achieves nearly perfect total-service proportional bandwidth sharing for competing parallel applications with diverse I/O patterns ($>$96% of the target sharing ratio). vPFS achieves 8.25 times better performance isolation than the native PVFS2, and its SFQ(D)+ scheduler achieves 3.35 times better performance isolation than the original

SFQ(D). It also makes efficient use of the storage and the SFQ(D)+ scheduler achieves 13.81 times better total throughput than a non-work-conserving scheduler.

CHAPTER 4

BIG-DATA STORAGE SYSTEMS MANAGEMENT

4.1 Introduction

Typical big-data computing systems are often built upon a highly scalable and available distributed file system. In particular, Google File System (GFS) [GGL03] and its open-source clone Hadoop Distributed File System (HDFS) [SKRC10] provide storage for massive amounts of data on a large number of nodes built with inexpensive commodity hardware while supporting fault tolerance at scale. A big-data application runs many tasks on these datanodes, which process the locally stored data in parallel via the I/O interface provided by such a distributed file system. In particular, the MapReduce programming model and associated run-time system are able to automatically execute user-specified map and reduce functions in parallel and handle job scheduling and fault tolerance [DG04]. Higher-level storage services such as databases (e.g., Hive [TSJ+10]) can be further built upon the distributed file system and provide more convenient interfaces (e.g., SQL) for users to process the data. Therefore, this dissertation focuses on big-data storage systems of the GFS/HDFS kind.

Although computing resources (CPUs) are relatively easy to partition, shared storage resources (I/O bandwidths) are difficult to allocate, particularly for data-intensive applications which compete fiercely for access to large volumes of data on the storage. Existing big-data systems lack the mechanisms to effectively manage shared storage I/O resources, and as a result, applications' performance degrades in unpredictable ways when there is I/O contention. For example, when one typical MapReduce application (WordCount) runs concurrently with a highly I/O-intensive application (TeraGen), WordCount is slowed down by up to 107%, compared to when it runs alone with the same number of CPUs.

I/O performance management is particularly challenging for big-data systems because of two important reasons. First, big-data applications have complex I/O phases (e.g., rounds of map and reduce tasks with different amounts of inputs, intermediate results, and outputs for a MapReduce application), which makes it difficult to understand their I/O demands and allocate I/O resources properly to meet their performance requirements. Second, a big-data application is highly distributed across many datanodes, which makes it difficult to coordinate the resource allocations across all the involved nodes needed by the data-parallel application. For example, the performance of a MapReduce application depends on the received total storage bandwidth from all the nodes assigned to its map and reduce tasks.

Both the map and reduce phases of a MapReduce application can spawn large numbers of map and reduce tasks on the GFS/HDFS nodes to process data in parallel. They often have complex but well-defined I/O phases. A *map* task is preferably scheduled to the node where its input data is stored. It reads the input from GFS/HDFS (either via the local file system or across the network) and *spills* and *merges* key-value pairs onto the local file system as intermediate results. A *reduce* task starts by *copying/shuffling* its inputs from all the map tasks' intermediate results (either stored locally or across the network). It then *merges* the copied inputs, performs the *reduce* processing, and generates final output to GFS/HDFS. Each of the above phases can have different bandwidth demands for input and output. Moreover, given the same volume of data to a map or reduce task, it can take different amount of time to process the data depending on the application's computational complexity.

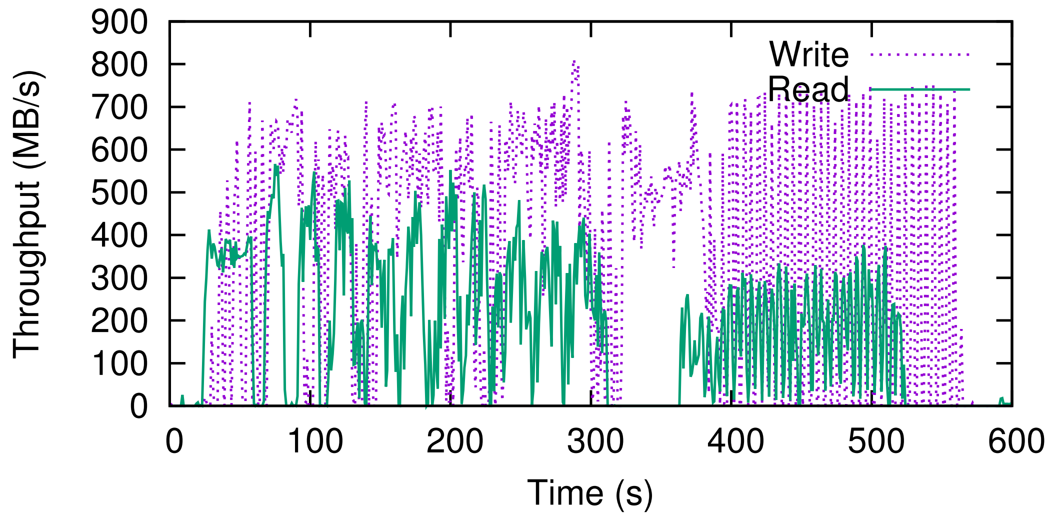
This dissertation proposes *IBIS*, an Interposed Big-data I/O Scheduler, to provide performance differentiation for competing applications' I/Os in a shared big-data system. This scheduler is designed to address the above-mentioned two challenges. First, *how to effectively differentiate I/Os from competing applications and allocate the shared stor-*

age bandwidth on the individual nodes of a big-data system? IBIS introduces a new I/O interposition layer upon the distributed file system in a big-data system, and is able to transparently intercept the I/Os from the various phases of applications and isolate and schedule them on every datanode of the system. IBIS also employs a new proportional-share I/O scheduler, SFQ(D2), which can automatically adapt I/O concurrency based on the storage load and achieve strong performance isolation with good resource utilization. Second, *how to efficiently coordinate the distributed I/O schedulers across datanodes and allocate the big-data system's total I/O service to the data-parallel applications?* IBIS provides a scalable coordination scheme for the distributed SFQ(D2) schedulers to efficiently coordinate their scheduling across the datanodes. The schedulers then adjust their local I/O scheduling based on the global I/O service distribution and allow the applications to proportionally share the entire system's total I/O service.

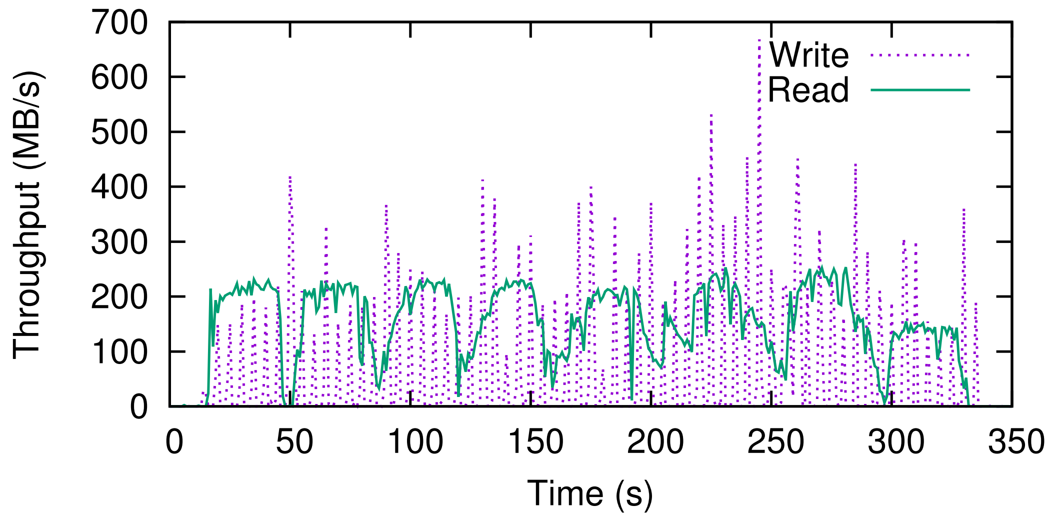
The rest of this chapter is organized as follows: Section 4.2 presents the need for big-data storage systems management; Section 4.3 introduces the virtualization approach of big-data storage systems; Section 4.4 presents the unique schduling algorithm to control I/O concurrency; Section 4.5 details the design of an efficient scheduler coordination scheme; Section 4.6 discusses the support for I/O management across different big-data frameworks; and Section 4.7 demonstrates the overhead, effectiveness and efficiency of IBIS solution implemented on YARN.

4.2 Motivation

The lack of I/O management in big-data systems presents a serious hurdle for data-intensive applications to get their desired performance. In a MapReduce system, on every single datanode, the tasks from different MapReduce applications compete with one another across all their phases for HDFS, local file system, and network I/Os. Across the

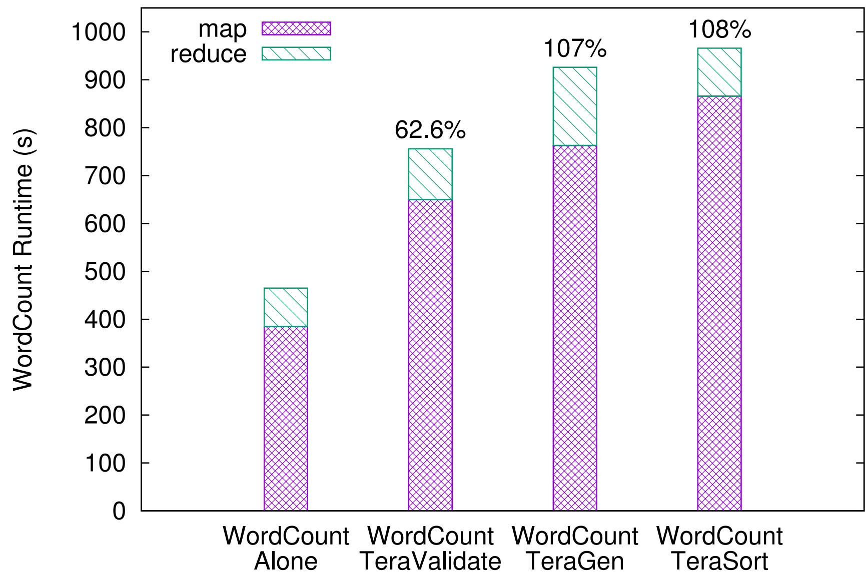


(a) TeraSort

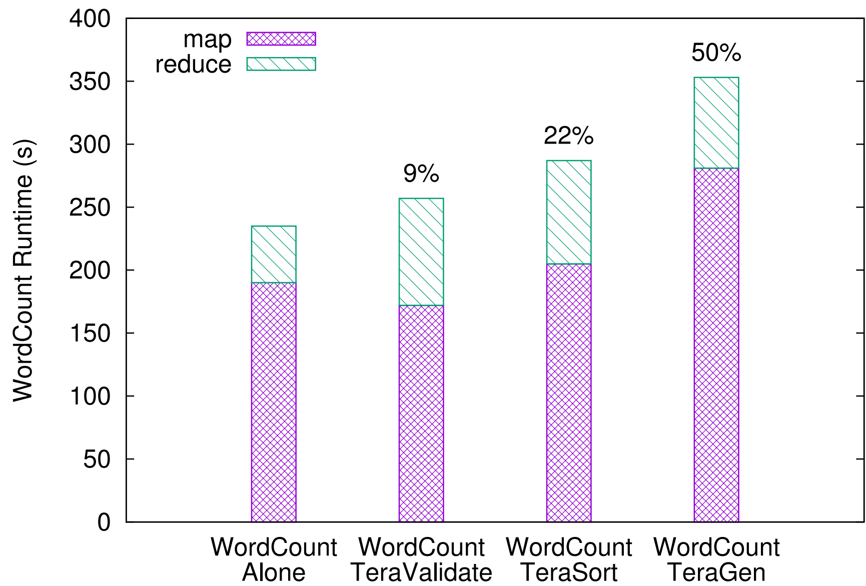


(b) WordCount

Figure 4.1: I/O demands of two classic MapReduce applications



(a) HDD setup



(b) SDD setup

Figure 4.2: WordCount performance slowdown

whole big-data system, these highly distributed applications also compete on many datanodes and their performance depends on the total amount of I/O services that they can get from all the involved nodes.

As an example of the diverse I/O demands of big-data applications, Figure 4.1 compares the I/O profile of two classic MapReduce applications, *TeraSort* and *WordCount*, each running alone with the same allocation of CPU and memory resources. These profiles show that *TeraSort* has a much more intensive I/O workload than *WordCount*. *TeraSort* has intensive HDFS reads and local file system writes in the map phase and intensive HDFS writes in the reduce phase. *WordCount*'s output is much smaller than its input, but there are plenty of intermediate writes throughout the map and reduce phases.

With such diverse big-data applications, the lack of I/O management will lead to severe and unpredictable performance interference between the applications. As an example of the I/O contention's performance impact, Figure 4.2 compares the performance of WordCount when it runs alone to when it runs with another application (TeraGen, TeraSort, TeraValidate) while keeping its CPU allocation (half of the 96 CPU cores in the system) the same. Details of the experiment setup are provided in Section 4.7. The results show substantial performance degradation in WordCount, which confirms the significant performance impact caused by I/O contention (CPU cache contention is relatively insignificant to the performance of these data-intensive applications). This dissertation addresses this serious problem with an interposed big-data I/O scheduling approach, IBIS, which is presented in the rest of this dissertation.

4.3 Interposed I/O Scheduling

The first question addressed by IBIS is *how to effectively differentiate the I/Os across the different phases of competing MapReduce applications on every datanode of a big-data*

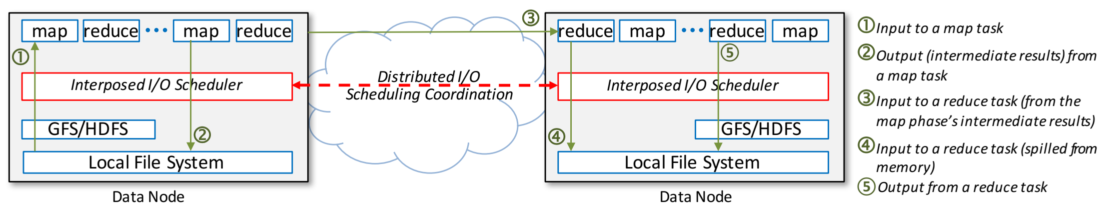


Figure 4.3: Achitecture of MapReduce and IBIS

system. The general design of IBIS is based on the *virtualization* principles, where an indirection layer exposes the interfaces already in use by the big-data system to access storage, allowing applications to time-share the storage system without modifications, while enforcing performance isolation and differentiation among them.

A key design decision that needs to be made in a virtualization approach is choosing the proper abstraction to introduce the virtualization layer. In the context of a big-data system, there are multiple layers in the storage hierarchy, from the applications, to HDFS, and to local file system and storage devices. On one hand, introducing virtualization at a higher layer can make use of more application knowledge to help the implementation, but it is more tied to specific applications and loses control of how I/Os are executed by the underlying layers. On the other hand, introducing virtualization at a lower layer of the storage hierarchy allows more control of I/O executions and can support more diverse applications, but it has to deal with more primitive I/O operations and loses application semantics that are useful for I/O differentiation.

Considering this tradeoff, IBIS is introduced upon the GFS/HDFS layer of the MapReduce storage architecture (Figure 4.3). This design can make effective use of application information to differentiate I/Os, because applications access the shared storage mostly through the HDFS interface. It also has enough low-level I/O control by scheduling the dispatch of I/Os to local file systems. Interposing of the applications' direct local file system I/Os and network I/Os are done at other interfaces at the same level. The rest of

this section details the interposition of these different types of I/Os used by a MapReduce application. All the modifications described below for implementing IBIS are made to Hadoop/YARN and do not require any change to applications.

Persistent I/Os are I/Os serviced by HDFS, where the inputs for map tasks are read from HDFS, and the outputs from reduce tasks are written to HDFS. Tasks use the *DFSClient* to interface with the *Data Node*, which represents an HDFS daemon, and Data Node converts the data requests, from both local and remote map tasks, to local file system I/Os. To differentiate I/Os from competing applications, the DFSClient interface is modified to carry application-specific information (job identifier and I/O service weight) as part of the header of each data request issued by the map/reduce tasks. These requests are scheduled by the IBIS component implemented in Data Node, which maintains a request queue for its local storage and dispatches the queued requests according to the chosen scheduling algorithm and policy.

Intermediate I/Os are I/Os to a datanode's local file system (not HDFS) for storing temporary data. Both map and reduce tasks use the local file system for spilling and merging in-progress data. The intermediate I/Os can also influence an application's performance. For example, a sorting program can generate the same amount of intermediate data as its input. In IBIS, these intermediate I/Os are first tagged with the job identifier and I/O service weight, and then routed to the IBIS component implemented within a local I/O scheduler, which can also reside in the Data Node daemon that runs on every datanode. IBIS schedules the intermediate I/Os in the same way as the persistent I/Os, following the same scheduling algorithm and policy.

Network I/Os occur during a shuffling phase between all the map tasks and reduce tasks. Because each reduce task's input is a partition of the map phase's outputs, it generally has to request a portion of the outputs from every map task. The data pulling thread launched by a reduce task is initiated with the job identifier and weight, which are car-

ried over in the header of the HTTP-based data requests. These requests are handled by the HTTP servlets which are implemented in the *Node Manager* daemons. Therefore, an IBIS scheduler is also implemented in the *Node Manager* to differentiate the network I/Os and schedule the corresponding local file system I/Os.

Note that IBIS does not rely on any bandwidth control from the network layer, and it is shown to be sufficient in the experiments because of two reasons: *1)* The storage is generally saturated before the network; *2)* By applying bandwidth control at the storage endpoints of the network I/Os, IBIS indirectly influences the contention on the network. However, IBIS can incorporate the network bandwidth control mechanisms such as OpenFlow [ope] if they are necessary and available, which will be left for future work.

In all the above I/O phases, concurrent requests from different applications are differentiated by their unique application IDs. An application obtains its ID from the job scheduler, which is carried over to all of its parallel tasks and used by the tasks to tag their I/Os for HDFS, intermediate, and network data. For every shared I/O service, these requests are queued and dispatched by an IBIS scheduler according the algorithm presented in the next section.

4.4 Proportional I/O Sharing

The second question addressed by IBIS is *how to allow the tasks from competing applications to proportionally share the I/O service of each datanode in a big-data system.* The interposed I/O scheduling framework in IBIS is flexible enough to support different algorithms. This dissertation focuses on algorithms that allow applications to proportionally share the I/O bandwidth, in the same way how they share the CPU time proportionally (e.g., using the Hadoop Fair Scheduler [fai]), so that it can provide the much needed, missing control knob for I/O allocation in big-data systems. Proportional resource sharing is

defined as when the total demand is greater than the available resource, each application should get a *share* of the resource proportionally to its assigned *weight*. Because only the relative values of weights matter to the bandwidth allocation, in the dissertation, the weight assignment to applications is often specified in terms of the *ratio* among the weights.

The proposed proportional-share scheduler is built upon the SFQ family of schedulers because of their computational efficiency, work-conserving nature, and theoretically provable fairness. SFQ schedules the backlogged requests from different applications using a priority queue, where each request's priority is positively affected by its application's weight and negatively affected by its cost (often estimated based on the size of the request). The scheduler can dispatch only one outstanding request, and it chooses the one with the earliest *start time* in the queue.

The SFQ(D) scheduler [JCK04] is an extension of SFQ for proportional sharing of storage resources which are commonly capable of handling multiple outstanding requests concurrently. The level of concurrency that the shared storage resource supports is captured by the *depth* parameter D in SFQ(D). The scheduler follows the original SFQ algorithm to dispatch queued requests, but it allows up to D outstanding I/Os to be serviced concurrently by the underlying storage in order to take advantage of the available I/O concurrency.

The choice of D has important implications on both fairness and resource utilization for a real storage system. On one hand, a larger D allows more concurrent I/Os and a higher utilization of the storage, but it may hurt fairness because of the scheduler's work-conserving nature. A more aggressive workload can use up all the storage bandwidth and even overload it, delaying the I/Os from a less aggressive workload. On the other hand, a smaller D gives the scheduler a tighter control on the amount of I/O share that a more aggressive workload can steal from others, and allows the I/Os from a less aggressive

workload to be serviced quickly when they arrive. It can thus improve fairness among the competing workloads but may lead to underutilization of the storage. So it is difficult to determine the optimal value of D statically, and it depends on the characteristics of the storage and workloads, some of which are also dynamic. It was in fact left as future work in the SFQ(D) paper [JCK04].

To address the above problem, this dissertation introduces a new SFQ-based algorithm, *Dynamic Depth SFQ*, or *SFQ(D2)* in short. It employs a feedback controller to automatically and dynamically adjust the value of D online. The controller works periodically (e.g., every second), and it decides the depth D_{k+1} for the next period $k+1$, based on the distance between the observed average I/O latency L_k of the previous period and the reference latency L_{ref}:

$$D_{k+1} = D_k + K_i \times (L_{ref} - L_k) \tag{4.1}$$

where K_i is an integral gain factor which determines how aggressively the controller works to reach the target latency. Following this equation, the controller automatically optimizes the value of D as it steers the observed I/O latency towards the reference latency.

The controller chooses I/O latency as the target because its goal is to maximize the storage utilization without compromising the fairness among applications, and I/O latency directly reflects the I/O performance of applications and the I/O load of the underlying storage. The reference latency is decided offline by profiling the storage using a synthetic MapReduce workload with increasing I/O concurrency. Both the I/O latency and throughput are measured during the profiling, and the I/O latency observed before the storage starts to saturate is the reference latency for the controller. Such profiling needs to be done only once for a given storage setup. If the storage's read and write performance are asymmetric such as in SSDs, the profiling can give separate reference latencies for reads and writes. In this case, the L_{ref} and L_k in the controller become the weighted av-

erage of the read latencies and write latencies, with the weights being the percentages of reads and writes observed in the previous control period.

This SFQ(D2) scheduler works upon the interposition layer described in Section 4.3 on every datanode of the big-data system. Each scheduler independently adjusts D based on its local dynamics in the workloads and underlying storage, and dispatches up to D number of outstanding I/Os from its local queue to the storage. This scheduler is used to provide proportional sharing of all the important I/O services offered by a datanode, including HDFS I/Os, temporary data I/Os, and network I/Os.

4.5 Distributed I/O Scheduling Coordination

The third question addressed by IBIS is *how to efficiently coordinate the distributed I/O schedulers across datanodes to support proportional sharing of a big-data system's total I/O service among competing applications.* A limitation of the IBIS scheduler described above is that a local scheduler's decision is made independently at each datanode, without accounting for information from other nodes. Local scheduling, based on only local knowledge, however, is not sufficient to deliver the desired performance differentiation from the perspective of the highly distributed big-data applications. The parallel nature of such an application requires it to get the necessary I/O service from all the nodes where its tasks are scheduled, and its performance depends on the total amount of I/O service that it gets from the system. Therefore, I/O management at the system level should support *total-service proportional sharing*, which means that the applications share the total I/O service from all the datanodes in the system proportionally to their assigned weights.

The challenge to achieving total-service proportional sharing is that applications often get unevenly distributed I/O services from the involved nodes. The exact amount of service that an application gets from a particular node depends on the number of CPU slots

that it gets on the node—which decides the I/O demands, and the applications running on the other slots of the same node—which decides the I/O contention. The number of slots that an application gets on a node in turn depends on the combination of, at any moment, the global CPU slot allocation policy, the application's data locality on the node, and the number of slots currently available on the node. Because of such uneven distribution of I/O services across the nodes, simply applying the same sharing ratio to each node and enforcing it using the local SFQ(D2) scheduler will not produce the same ratio of sharing of the total I/O service.

To address this challenge, IBIS enables the distributed SFQ(D2) schedulers to coordinate with one another and enforce total-service proportional sharing collaboratively. Every scheduler shares its local I/O service distribution—the applications that it serves and the amounts of services that they get locally, with the other schedulers. Based on the global I/O service distribution, every scheduler can then adjust its local I/O service distribution so that the total services that the applications get are proportionally to their assigned weights. Specifically, IBIS follows the algorithm in DSFQ [WM07] to adjust local SFQ scheduling for total-service proportional sharing. When an SFQ(D) scheduler considers the scheduling of a queued request, it delays the request's *start time* by the total amount of service that the corresponding application has received from all the other nodes. In this way, the local scheduler dispatches the requests from different applications according to their received total I/O services, not just the local services.

Another challenge that must be addressed by IBIS is how to efficiently coordinate a large number of distributed schedulers in a big-data system. If every scheduler has to broadcast its information to all the other schedulers, it can easily overwhelm the schedulers and the network as the system scales out. DSFQ [WM07] work assumes a traditional remote I/O model, where the clients send their I/Os to remote storage nodes and a coordinator can be interposed in between to gather and pass on the global I/O service

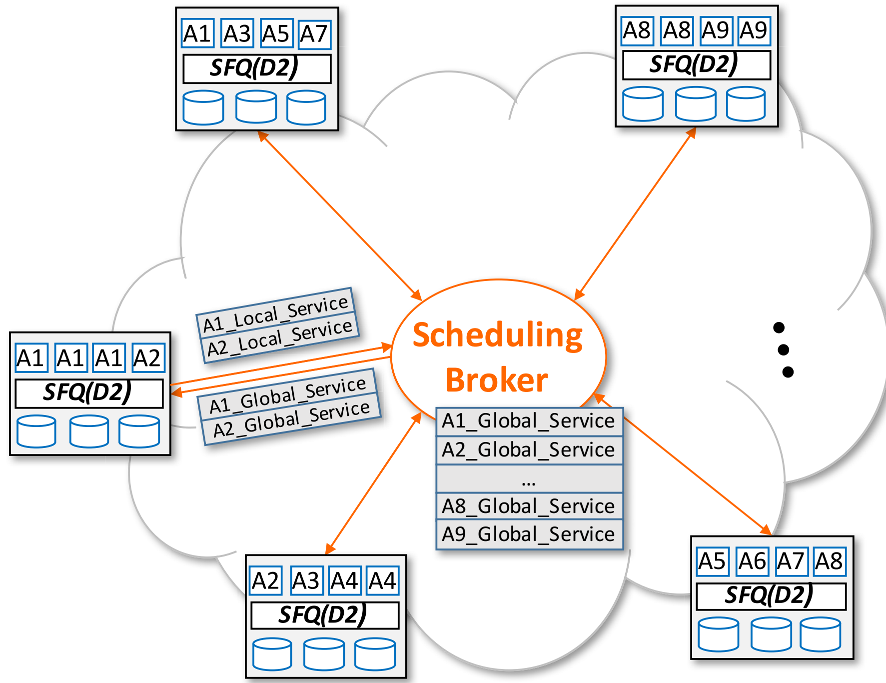


Figure 4.4: Architecture for distributed I/O scheduling coordination

distribution. But this approach does not apply to a big-data system, where computing tasks are shipped to the nodes where their data is stored and they process the data using primarily local I/Os.

To solve this problem, IBIS employs a centralized *Scheduling Broker* to facilitate the information exchange among the distributed schedulers in a scalable manner (Figure 4.4). Every local scheduler $j \in \{1,\ldots,m\}$ sends its current I/O service distribution—a vector of *local* I/O service amount a_{ij} for each application $i \in \{1,\ldots,n\}$ that the scheduler j serves—to the broker periodically (e.g., every 1 second). Based on the information received from all the local schedulers, the broker summarizes the total I/O service $A_i = \sum_{j=1}^{m} a_{ij}$ for each application i in the system. It then responds to a local scheduler's message with the total I/O service distributions—a vector of *total* I/O service amount A_i

for each application *i* that the local scheduler currently serves. Based on this total service information, the local scheduler can then adjust its scheduling as discussed above.

The overhead of this scheduling coordination scheme is small. The size of the messages between a local scheduler and the broker is bounded by the number of applications that the scheduler currently serves. The state that the broker needs to maintain is simply a vector of total I/O service amount for all the applications currently in the system. The frequency of coordination can be adjusted based on the desired granularity of fairness and the scale of the system—more frequent coordination reduces transient unfairness but increases the overhead; and vice versa. Hadoop/YARN already employs centralized managers, in particular the *Resource Manager* for coordinating the distributed *Node Managers*, which is shown to be scalable for managing thousands of nodes [VMD+13]. In fact, in the IBIS implementation, the I/O Scheduling Broker is embedded as part of the Resource Manager and the I/O scheduling coordination information is piggybacked on the existing communications between the managers to further reduce its overhead.

4.6 Multi-framework I/O Scheduling

Big-data resources are increasingly shared by diverse computing frameworks [DG04, TSJ+10], as users have different data processing requirements as well as different preferences of programming models. No single framework is perfect for all big-data problems and all users. Solutions such as YARN [VMD+13] and Mesos [HKZ+11] allow different frameworks to share the same set of resources and employ mechanisms such as containers [con] to allocate CPU cores and memory capacity to the resource-sharing applications. However, these resource management solutions still cannot provide strong performance isolation, because they do not support the allocation of shared I/O resources which the data-intensive applications have to compete for. As the experiments will show in Sec-

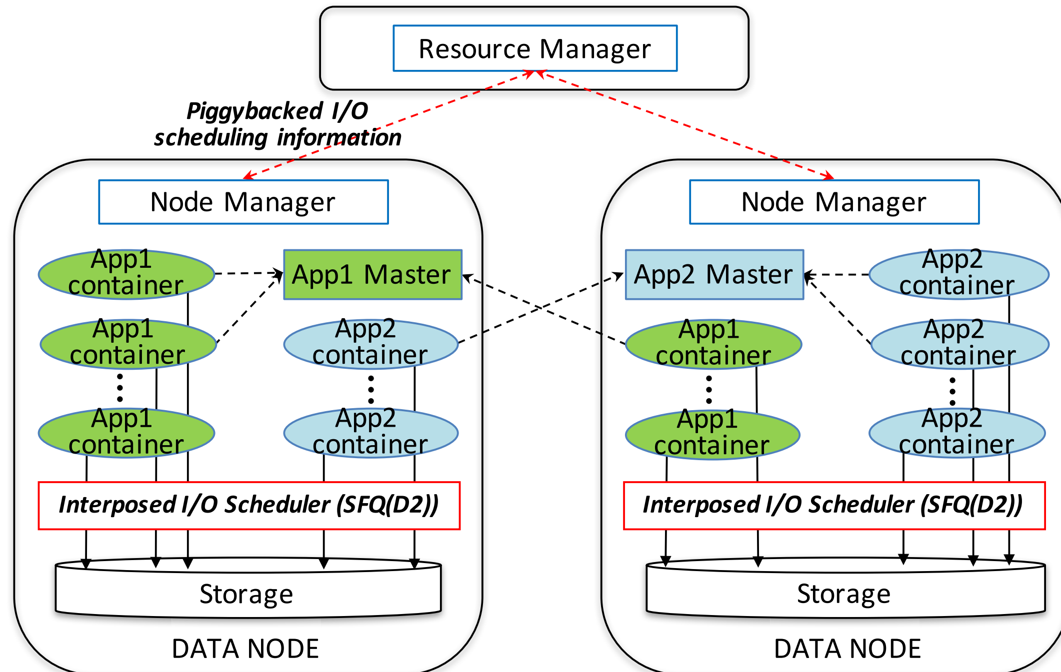


Figure 4.5: Integration of IBIS with YARN

tion 4.7.4, although containers do provide some level of I/O isolation, it is not sufficient. Containers can control only the I/Os directly issued to the local file system, e.g., intermediate I/Os from MapReduce, but not the distributed I/Os, e.g., HDFS I/Os, which are serviced by a shared datanode server and cannot be differentiated using the container mechanism.

Thus, existing multi-framework resource management solutions still need IBIS to provide the missing I/O control knob for effective I/O bandwidth allocation. Specifically, in YARN, IBIS is seamlessly integrated in its *Application Master*, *Resource Manager*, *Node Manager*, and *Data Node* components (Figure 4.5).

IBIS allows an application to specify its required total I/O bandwidth (e.g., 300MB/s) to its *Application Master*, in addition to the amount of required CPUs and memory (e.g., 64 CPU cores and 64GB RAM), in order to achieve its desired performance. The centralized *Resource Manager* collects the resource requests from the concurrent *Application*

Masters and uses IBIS to determine the global, total-service I/O bandwidth allocation, as well as allocating the CPUs and memory using an existing scheduler such as the Fair Scheduler. The *Resource Manager* then coordinates with the distributed *Node Managers* to enforce the resource allocations on every datanode. Each *Node Manager* uses the local *Data Node* to schedule the local I/Os according to the global, total-service I/O sharing target, similarly to how it uses containers to enforce the CPU and memory allocations to the local data processing tasks.

Finally, the IBIS scheduler in *Data Node* interpose all the I/Os, as discussed in Section 4.3 and uses SFQ(D2) discussed in Section 4.4 to schedule the I/Os according to the given bandwidth allocation.

4.7 Experimental Evaluation

4.7.1 Setup

The experimental evaluation was done on a cluster of nine nodes each with two six-core 2.4GHz AMD Opteron CPUs, 32GB of RAM, and two 500GB 7.2K RPM SAS disks, interconnected by a Gigabit Ethernet switch. All the nodes run the Debian 4.3.5-4 Linux with the 3.2.20-amd64 kernel and use EXT3 as the local file system. The evaluation was performed in YARN 2.7.0 with the IBIS prototype implemented in the *Resource Manager*, *Node Manager*, *Application Master*, and *Data Node* as described in Sections 4.3 and 4.6. Eight nodes are dedicated to run applications consuming up to 96 CPU cores and 192GB memory by their tasks, where each map task uses 1 CPU core and 2GB of memory and each reduce task uses 1 CPU core and 8GB of memory. One additional node runs the YARN *Resource Manager* and *Name Node* and the IBIS scheduling broker. The two disks on each node are used to store HDFS data and intermediate data separately. The

Table 4.1: The YARN configuration used in the evaluation

Key	Value
dfs.replication	3
dfs.block.size	134217728
fairscheduler.preemption	true, 5s

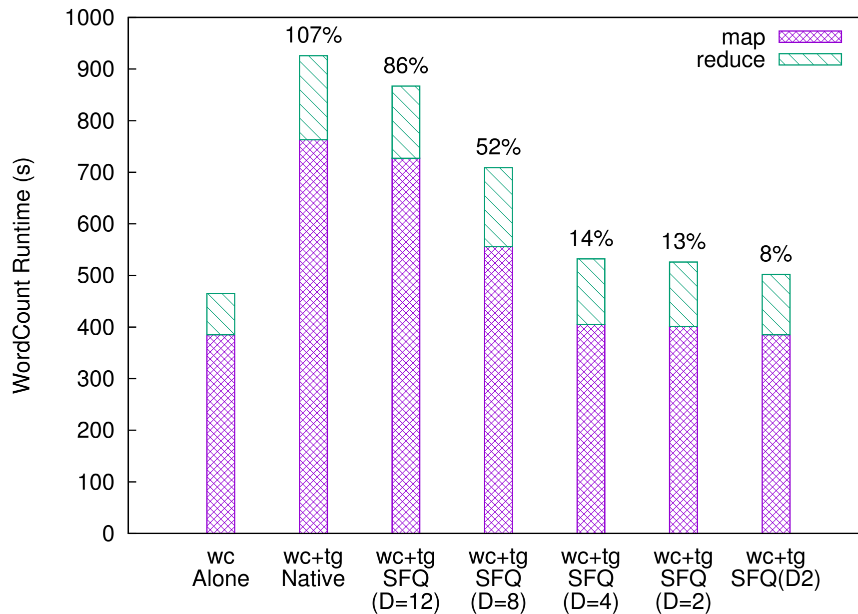
configuration parameters of YARN and its Fair Scheduler used in the evaluation are listed in Table 4.1.

The evaluation compares the performance of IBIS to native Hadoop with YARN using a variety of benchmarks, including TeraGen (1TB output), TeraSort (50–400GB input), WordCount (50GB Wikipedia input), Facebook2009 [SWI], and TPC-H on Hive [TSJ+10] (53GB input), which are explained in detail in the following experiments. For IBIS with the SFQ(D2) scheduler, the control period is set to 1 second.

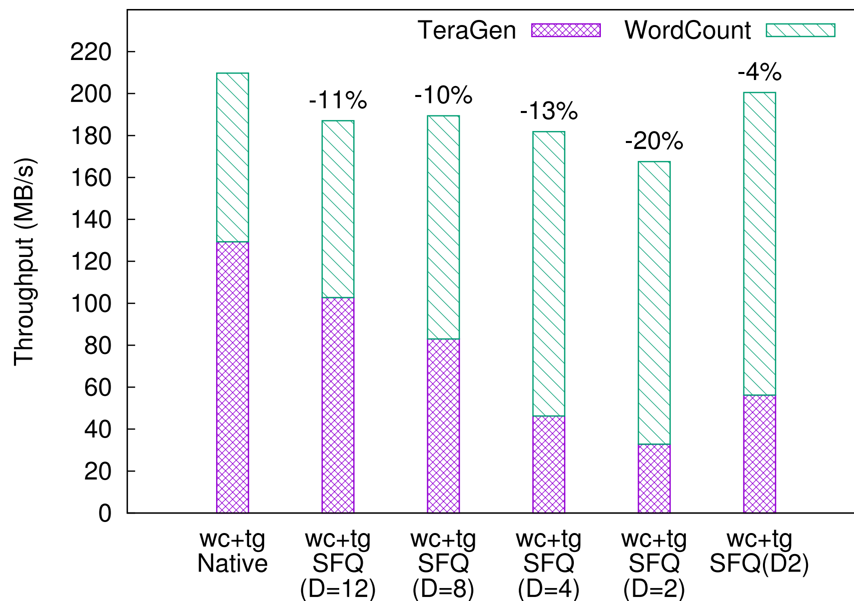
4.7.2 Performance Isolation (WordCount)

The first experiment evaluates whether IBIS is able to provide performance isolation to one application while it is under intensive I/O contention from others as in the motivating example discussed in Section 4.2. It is an important policy in many scenarios where the performance of an important big-data application must be guaranteed regardless of the contention from others. As in the motivating example, Figure 4.6a shows that when WordCount runs with TeraGen, it is slowed down by 107% due to I/O contention, compared to when it runs alone with the same CPU allocation (48 map slots and 16 reduce slots). Performance isolation is challenging to accomplish for WordCount because its I/O rate is much lower than TeraGen, while a work-conserving I/O scheduler tries not to under-utilize the storage.

Figure 4.6a shows the results of IBIS from using both the original SFQ(D) scheduler with a static value of *D* and the new SFQ(D2) scheduler which dynamically adjusts *D*.



(a) Runtime of WordCount. The numbers on top of the bars are the slowdown w.r.t. the standalone runtime. The shuffling time of the first wave of reduce tasks is overlapped with the map phase and not shown in the bars. But the height of the bars reflects the total runtime.



(b) Total throughput of WordCount and TeraGen.

Figure 4.6: WordCount performance slowdown with HDDs

The sharing ratio between WordCount and TeraGen is set to 32:1 to favor WordCount, but TeraGen can always use the spare I/O bandwidth because the schedulers are work-conserving. Comparing the results from SFQ(D) with different *D* values, it shows that reducing *D* does give the scheduler a tighter control on I/O scheduling and achieves better performance isolation for WordCount, reducing its slowdown to as low as 13%. Comparing the results from SFQ(D) to SFQ(D2), it shows that the new scheduler achieves the best isolation for WordCount with a runtime that is only 8% slower than when it runs alone, and it does so by automatically adjusting the value of *D*.

Note that the 32:1 sharing ratio is used here because the objective of this experiment is to restore the performance of WordCount without underutilizing the bandwidth. Lower sharing ratios would favor WordCount less and result in worse performance of WordCount while still being much better than the native case. For example, a sharing ratio of 2:1 restores WordCount's performance to 148% of its standalone runtime with SFQ(D=2) and 118% with SFQ(D2).

The excellent performance isolation from IBIS is accomplished while still allowing the competing application, TeraGen, to make good progress and fully utilize the underlying storage. To confirm this, Figure 4.6b compares the total throughput of WordCount and TeraGen when they run on native Hadoop without I/O management vs. when they run on IBIS. The native case has the highest total throughput, because TeraGen's I/Os are sent to storage as soon as they come without any control. In comparison, the number of outstanding I/Os is controlled by *D* in the schedulers of IBIS. The results show that IBIS can achieve good storage utilization in all configurations, where the best result is also from SFQ(D2) which is only 4% lower than the native case. This result is achieved while reducing WordCount's runtime slowdown from 107% to 8% as discussed above.

To provide a detailed view of how SFQ(D2) works, Figure 4.7 shows how it adapts *D* based on the observed I/O latency on one of the datanodes. It follows the equation for

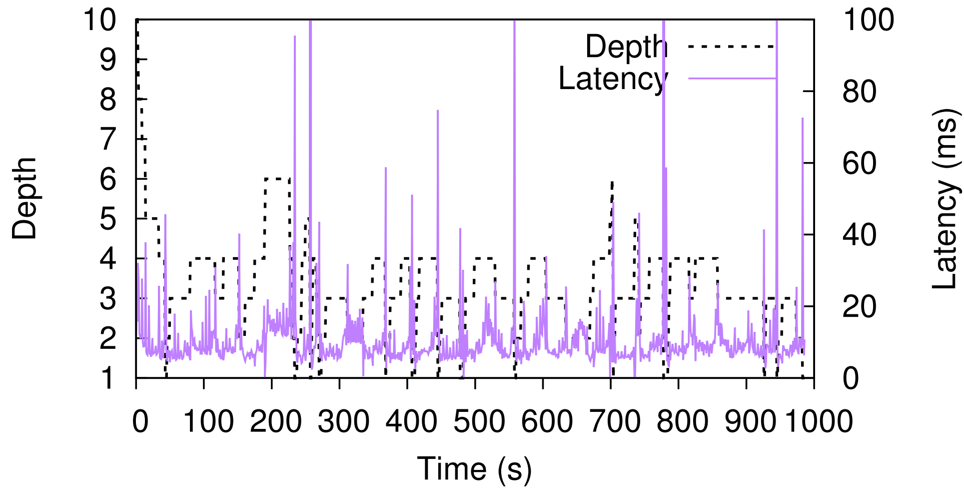
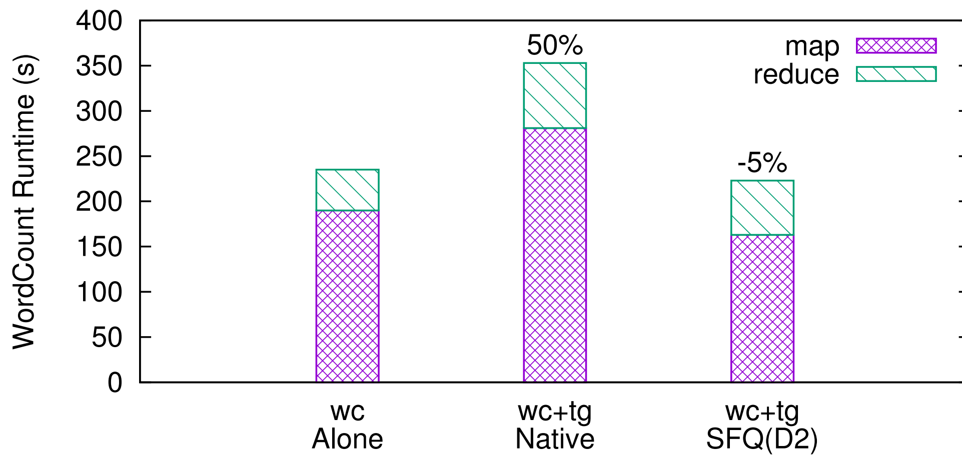


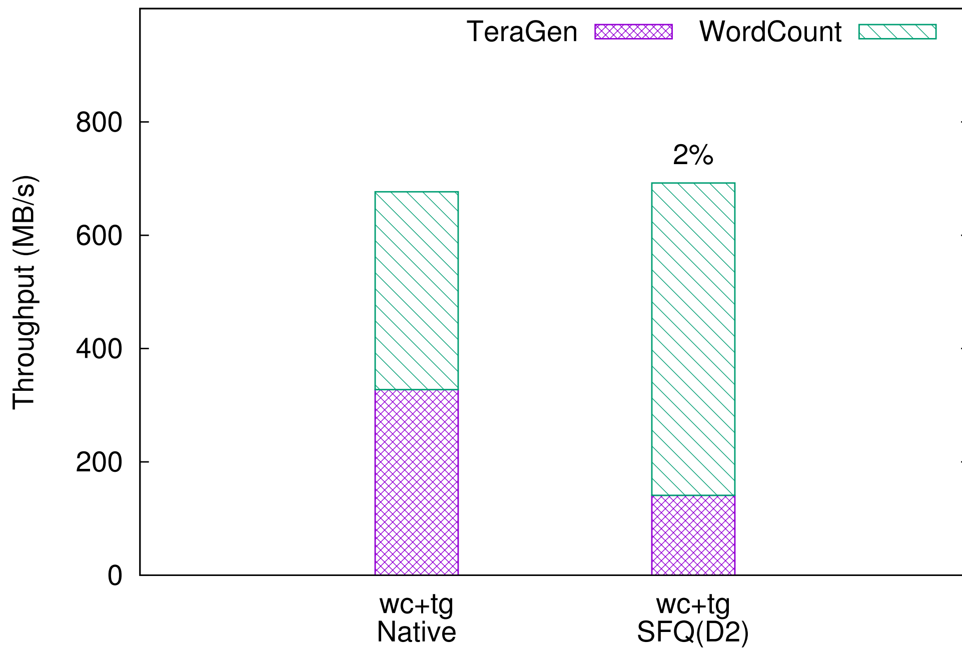
Figure 4.7: Adaptation of D by SFQ(D2)

the feedback controller described in Section 4.4. The gain factor is set to 10^{-6}. The value of D is bounded between 1 and 12. Throughout the run the controller reacts quickly to the observed latency and adapts D quickly to sustain strong performance isolation with good resource utilization. Noticeable that at the 260th second and 790th second, the underlying storage system undergoes foreground flushing of writes buffered in memory and causes the high spikes in I/O latency, while the controller still responds in a timely manner. Although IBIS does not have direct control of such lower-level dynamics, it can still effectively mitigate their impact by timely adapting the I/O concurrency. It is therefore able to sustain good performance isolation without having to modify the underlying storage layers which would be much more intrusive and expensive.

Although faster storage devices such as SSDs are increasingly considered by big-data systems, they cannot completely replace HDDs due to their limited capacity. Moreover, faster storage does not make the I/O contention problem go away; the increasing volume and velocity of big-data will always demand I/O management. To confirm this, the same experiment is repeated on a different storage setup using SSDs (Intel 120GB MLC SATA-interfaced flash devices) to store both HDFS and temporary data on each datanode. The



(a) Runtime of WordCount



(b) Total throughput of WordCount and TeraGen

Figure 4.8: WordCount performance slowdown with SDDs

results in Figure 4.8a first confirm that WordCount is still severely interfered (50% slowdown) by TeraGen on native Hadoop due to I/O contention. They also confirm that IBIS still achieves strong performance isolation with excellent storage utilization for this faster storage setup. Interestingly, IBIS with SFQ(D2) achieves a better runtime for WordCount than when it runs alone, and a better total throughput for WordCount and TeraGen than native Hadoop. This can be explained by the read/write asymmetry of flash devices and the implicit promotion of reads in SFQ(D2). Writes are much slower than reads on flash devices and they can significantly slow down the reads that are scheduled after them. When intensive writes are received by the scheduler, it automatically reduces D, which gives the reads a better chance to establish backlogged requests and be dispatched before some of the writes, therefore achieving better overall performance. This unique characteristic of flash devices will be further studied in the future work to optimize the IBIS scheduler specifically for the use of SSDs in big-data systems.

4.7.3 Performance Isolation (Facebook2009)

The second experiment evaluates whether IBIS is also able to provide performance isolation to the Facebook2009 workload, which is far more diverse than WordCount. A total of 50 jobs are created using the SWIM workload generator [SWI], by sampling the historical Facebook job logs and emulating their computing and I/O phases. The samples are down-scaled to fit the size of this dissertation's testbed. The workload consists of diverse MapReduce applications, including both small and large jobs and having different levels of I/O demands, with their input-to-shuffle ratio and shuffle-to-output ratio varying between 0.05 to 10^3 and 2^{-5} to 10^2 respectively. These ratios represent the relative dataset sizes between map input to shuffle input and between shuffle input to reduce output. Vary-

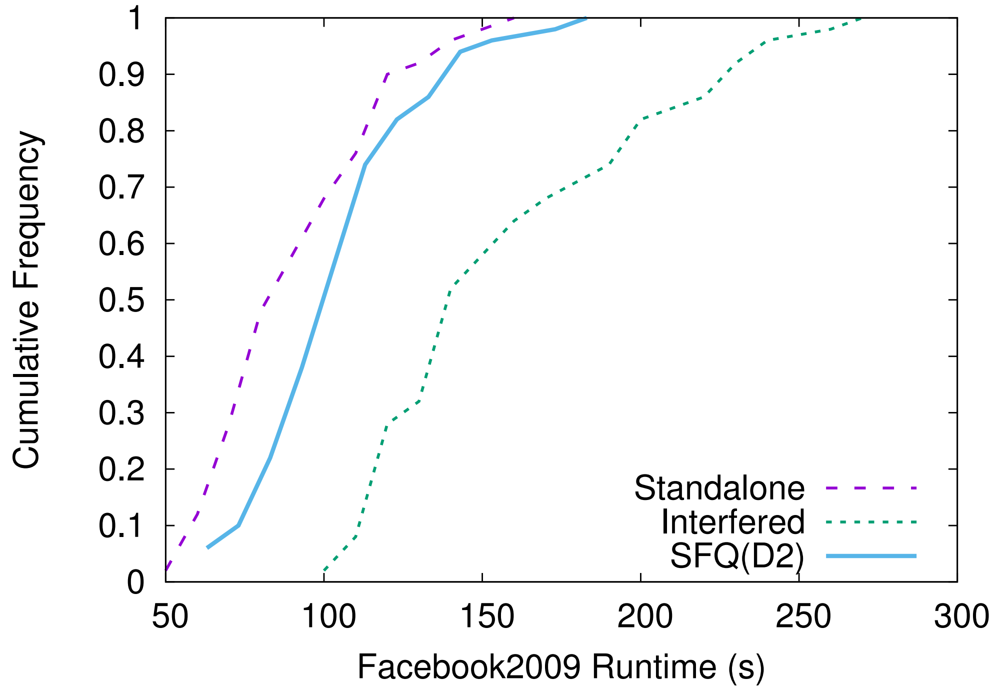


Figure 4.9: Cumulative distribution of Facebook2009 job runtimes

ing these ratios therefore generates different levels of computation and I/O intensities for the various phases of the jobs.

The Facebook2009 jobs are run together with TeraGen on the native Hadoop (*Interfered*) and on IBIS using the SFQ(D2) scheduler with a bandwidth sharing ratio of 32:1 favoring Facebook jobs (*SFQ(D2)*). As a baseline, Facebook2009 is also run alone without I/O contention from others (*Standalone*). The CPU and memory resources allocated to Facebook2009 are kept to half of the total resources for all the cases.

Figure 4.9 compares the cumulative distribution of the Facebook2009 jobs' runtimes. In the *Standalone* case, 90% of Facebook2009 jobs finish within 120s. When they run together with TeraGen without I/O management in the *Interfered* case, they are impacted drastically by TeraGen, and no job finishes within 50s and 90% of them take up to 230s. In comparison, using *SFQ(D2)*, IBIS is indeed able to provide strong isolation to Facebook2009, and 90% of the jobs can finish within 138s. Comparing the average runtime

of Facebook2009, it is reduced from 168s in the *Interfered* case to 115s under *SFQ(D2)*, where the *Standalone* average runtime is 98s. Most of these jobs require only one wave of map and reduce tasks. Without an I/O scheduler, their I/Os are severely interfered by TeraGen and slowed down substantially. With IBIS, they are well isolated from TeraGen and can utilize the allocated storage bandwidth to achieve a performance close to the standalone case.

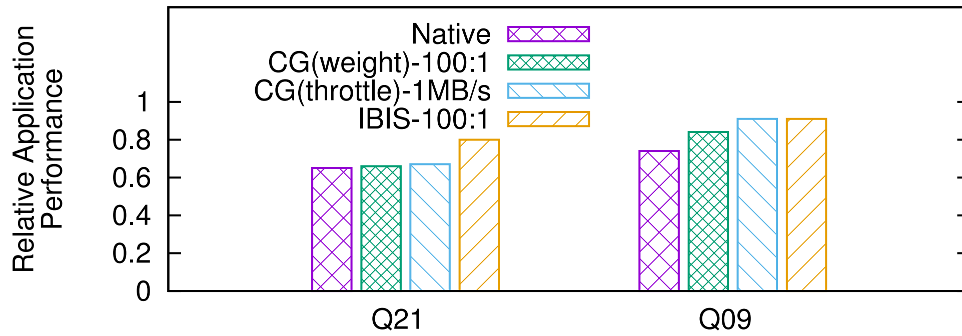
4.7.4 Multi-framework I/O Scheduling

The third experiment evaluates IBIS' ability to schedule I/Os and manage their performance for different big-data frameworks, Hive [TSJ+10] and MapReduce, that share the same infrastructure. Specifically, this experiment considers TPC-H queries [tpc] as the benchmark for Hive. TPC-H represents decision support systems scanning large volumes of business data, executing queries with a high degree of complexity, and providing keys to important business questions. Hive is a data warehouse framework built upon Hadoop, to support the SQL query execution for data stored on HDFS. Its execution engine spawns a series of MapReduce jobs for query fulfillment, providing end users with much flexibility in data format adaptation and ease of use in a scalable cluster environment.

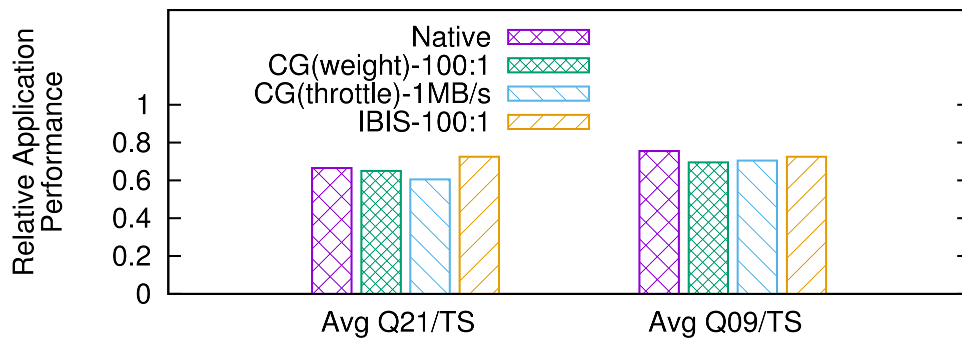
The experiment focuses on the TPC-H queries Q9 (*product type profit*) and Q21 (*suppliers who kept orders waiting*) which involve multiple intensive I/O phases including both HDFS and intermediate I/Os. Q9 reads 53GB of initial input from five tables stored on HDFS and generates 120GB of intermediate I/Os. Q21 reads 45GB of initial input from four tables on HDFS, and generates 40GB of intermediate I/Os. Both queries launch up to 15 sequential Hadoop jobs. Q9's final output is 5KB and Q21's final output is 2.6GB.

The TPC-H queries on Hive and TeraSort on MapReduce are run concurrently, each with half of the CPU cores and memory. Although the native YARN does not provide any support for I/O management, it is conceivable to extend it to use cgroups [con], which YARN already uses to allocate CPUs and memory, to also manage I/O bandwidth allocation. To compare to this cgroups-based approach, YARN is extended to use the cgroups mechanisms to allocate shared I/O bandwidth between the two frameworks. This extended YARN can use both the proportional-sharing and throttling modes of cgroups to manage I/Os. In the proportional-sharing mode, the shared bandwidth is allocated to competing applications according to their assigned weights. In the throttling mode, a specific cap can be set to an application's bandwidth usage. Note that as discussed in Section 4.6, this approach as well as other similar ones can manage only the intermediate I/Os, but not HDFS I/Os; in contrast, IBIS is able to differentiate both local and distributed I/Os and schedule them according to the given performance policy.

Figure 4.10a shows the relative performance of the two TPC-H queries when running against TeraSort w.r.t. their standalone runtimes. For Q21, on *Native* YARN, the query experiences a 35.2% performance loss when compared to its standalone runtime. When using cgroups' two different modes with aggressive parameters to favor TPC-H—100:1 bandwidth sharing ratio in *CG weighted 100:1* and 1MB/s bandwidth cap on TeraSort in *CG throttled 1MB/s*, it can only improve the query performance by 1.2% and 2.5% respectively. In comparison, IBIS is able to improve the query performance to within 80% of its standalone performance, which is 15.2% better than native YARN and 12.7% better than cgroups. For Q9, the query experiences a 26% performance loss when running against TeraSort on *Native* YARN. Both cgroups' throttling policy and IBIS can restore the query performance to 91% of its standalone runtime, which is better than cgroups' proportional-share policy by 8%. The cgroups-based I/O throttling works better for Q9 than Q21, because Q9 has a higher level of intermediate I/Os which can be throttled by



(a) Performance of TPC-H queries (Q9 and Q21) relative to their standalone runtimes



(b) The average relative performance of TPC-H and TeraSort

Figure 4.10: Performance restoration of TPC-H on Hive

cgroups. However, throttling causes underutilization of storage and unnecessary slowdown of the competing application, TeraSort. Consequently, the performance of TeraSort is up to 16% worse when using cgroups throttling, compared to IBIS which is work-conserving.

To evaluate the overall system performance considering both competing frameworks, the experiment considers the *average relative performance* of the two applications, i.e., the average of each application's relative performance w.r.t. its own standalone performance. Figure 4.10b shows that when Q21 runs with TeraSort, the two applications experience a 26% performance loss in average on *Native* YARN, and the use of cgroups-based proportional bandwidth sharing does not improve it. The I/O throttling policy of cgroups makes it even worse because it is non-work-conserving and causes underutilization of the I/O bandwidth. In comparison, IBIS is able to achieve an average relative performance of 80%. For Q9, cgroups and IBIS achieve similar average relative performance, which is about 4% lower than *Native* because this query is more I/O intensive and incurs a higher overhead in I/O scheduling.

Considering the results from both figures, a multi-framework resource management solution such as YARN cannot provide strong performance isolation among competing applications due to the lack of I/O management. In comparison, IBIS is able to provide I/O isolation and when used in combination with YARN's CPU and memory management, it is able to protect the performance of a vulnerable application such as TPC-H while still allowing the competing, intensive application such as TeraSort to make good progress by using the available storage bandwidth.

4.7.5 Proportional Slowdown

The previous three experiments are designed to show IBIS' ability to support the performance isolation policy. Another important and commonly used policy is *proportional slowdown*, i.e., the relative performance of competing applications, w.r.t. their standalone performance, is proportional to their assigned weights. This policy is often used to achieve fairness for applications in terms of their performance, not their resource allocations. A big-data application's performance depends on both the available CPU cores and I/O bandwidth, and its use of CPU and I/O resources are correlated. Without control on the I/O bandwidth, it is possible to achieve proportional slowdown by limiting the CPU slots allocated to the more I/O-intensive application and indirectly throttling its I/O rate, so that the less I/O-intensive one can get more I/O bandwidth. Nonetheless, such a configuration leads to under-utilization of the storage and suboptimal performance of the applications.

With IBIS, system administrators can tune both CPU slot and I/O bandwidth allocations together, and achieve proportional slowdown without wasting the resources. Ideally, this tuning should be done automatically without human intervention, which would require performance models of the big-data applications that can capture their CPU and I/O resource demands given different performance targets. How to create such models and use them to automatically tune the resource allocations are interesting research problems on their own and will be considered in the future work. This dissertation focuses on the problem of providing the necessary I/O control mechanisms to support a variety of performance policies such as performance isolation and proportional slowdown, which comes with a set of unique challenges discussed earlier and is tackled by the proposed IBIS framework. Without such control knobs enabled by IBIS, it would be difficult for either administrators or autonomic software to achieve the desired performance policy with efficient resource utilization.

In this experiment, *equal slowdown* of both TeraSort and TeraGen is the target policy, meaning that both applications should be slowed down by the same percentage relative to their respective standalone runtime. Figure 4.11 shows the performance slowdown of these two applicationsi. By adjusting only the CPU allocation using the Hadoop Fair Scheduler, the best equal slowdown that it can get is 83% slowdown for TeraSort and 61% for TeraGen. By using Fair Scheduler and IBIS to tune both CPU and I/O allocations together, it is able to get a perfect equal slowdown of 42%, which is 30% better than the average slowdown of them when using Fair Scheduler only. These results therefore confirm that IBIS is also able to support the proportional slowdown policy and optimize the application performance under this policy.

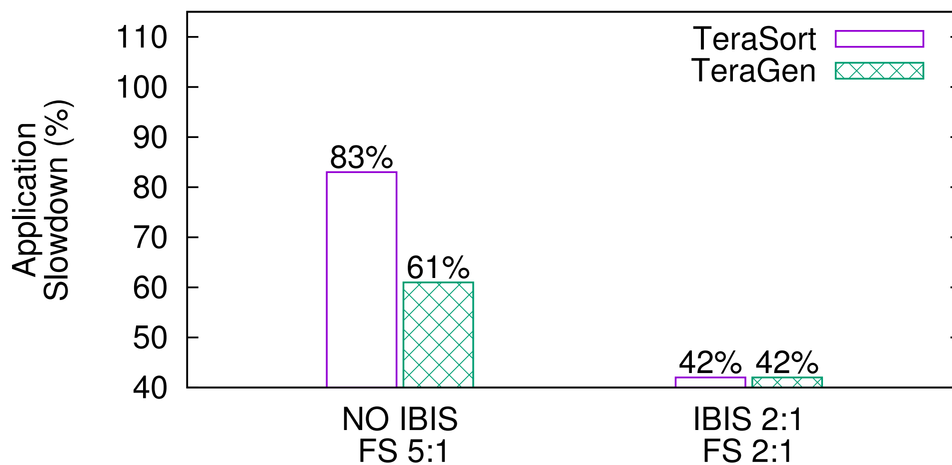


Figure 4.11: Performance slowdown of TeraSort and TeraGen with or without IBIS

4.7.6 Coordinated Scheduling

As discussed in Section 4.5, many factors decide the I/O service that an application gets from each storage node in a big-data system, including data distribution, slot allocation, task assignment, and competing applications, which all contribute to the uneven distribution of I/O services across the nodes. Without a mechanism for coordinating the dis-

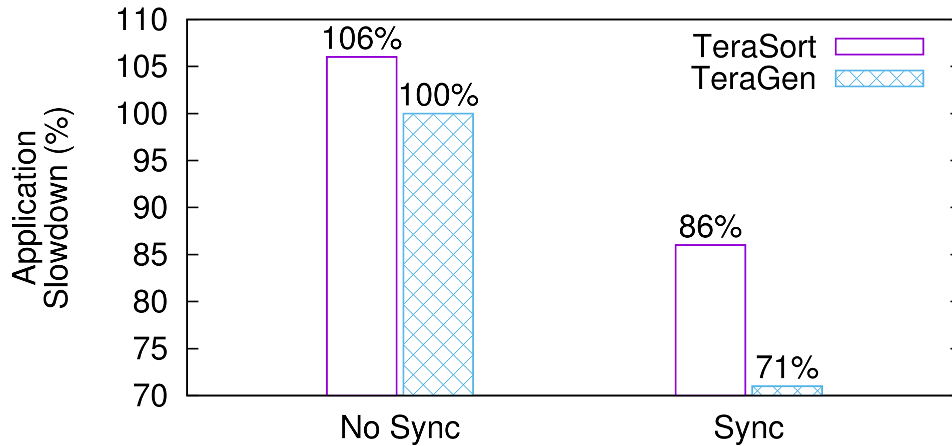


Figure 4.12: Performance slowdown of TeraSort and TeraGen with or without sync

tributed I/O schedulers and an algorithm to adjust local sharing ratios based on the global sharing policy, the total service that an application gets from the entire big-data system will diverge from the given target. This experiment evaluates the effectiveness of the distributed scheduling coordination mechanisms (Section 4.5) for achieving total-service proportional sharing.

The experiment is conducted similarly to the previous one for achieving equal slowdown for TeraSort and TeraGen, but it considers two different IBIS setups where the distributed scheduling coordination is disabled (*No Sync*) and enabled (*Sync*). The latter case should allow IBIS to find better equal slowdown because it can dynamically adjust local I/O service distribution based on global service distribution, which leads to better CPU and I/O resource utilization and better performance for both applications. Figure 4.12 shows the performance slowdown of TeraSort and TeraGen with respect to their own standalone runtimes. The average performance slowdown with *Sync* is 25% better than from *No Sync*, confirming the improvement made by the coordinated I/O scheduling.

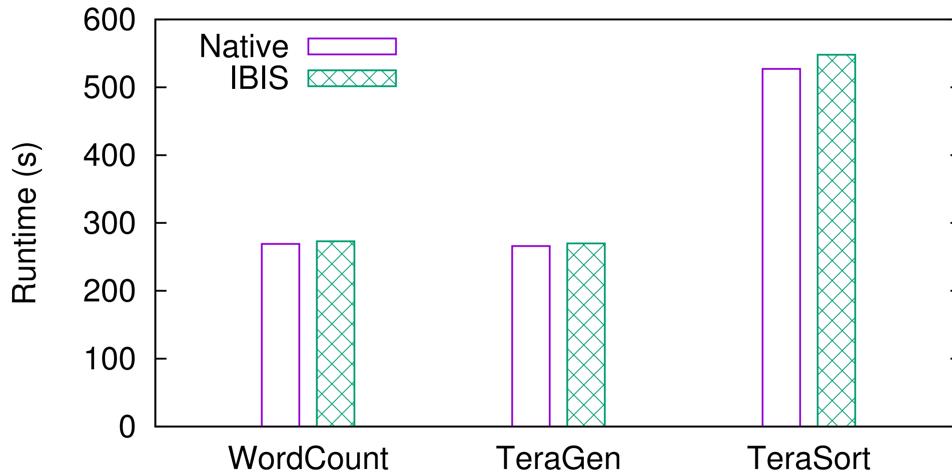


Figure 4.13: Runtime overhead of IBIS

Table 4.2: CPU and memory usages of the YARN and IBIS daemons including the Resource Manager, Node Manager, and Data Node

Benchmark	Resource	Native	IBIS
WordCount	CPU	0.4%	0.5%
TeraGen	CPU	1.7%	5.1%
TeraSort	CPU	0.55%	0.65%
WordCount	Memory	1.2%	8.2%
TeraGen	Memory	2.0%	8.1%
TeraSort	Memory	1.6%	10.6%

4.7.7 Overhead

The last experiment evaluates the overhead of IBIS from several aspects. First, it studies the performance impact to a big-data application from IBIS-based I/O interposition and scheduling. WordCount, TeraGen, and TeraSort are all considered because each of them has distinct I/O patterns and demands. They are run separately with all the 96 CPU cores in the system. Figure 4.13 shows that the overhead of using IBIS is 1%, 2%, and 4% for WordCount, TeraGen, and TeraSort, respectively, in terms of runtime.

Second, the resource usages of IBIS are evaluated by tracking the total CPU and memory utilizations of the Hadoop TaskTracker, DataNode, and JobTracker, where the

Table 4.3: Development cost of IBIS

Component	**Lines of Code**
Interposition	2593
SFQ(D) Scheduler	734
SFQ(D2) Scheduler	1520
Scheduling Coordination	1705
Total	6552

IBIS implementation is located. Table 4.2 lists the per-core CPU utilization and per-node memory utilization, which are reasonable compared to native Hadoop's resource usages.

Third, Table 4.3 summarizes the code development complexity in terms of lines of code categorized by the IBIS components. IBIS provides a flexible big-data I/O scheduling framework, and allows users to conveniently create new schedulers for different objectives. The amount of work required to develop a sophisticated scheduler on IBIS is only at the level of a thousand lines of code.

4.8 Summary

IBIS is an Interposed Big-data I/O Scheduler to address the challenges and provide the much needed I/O performance differentiation to diverse big-data applications, possibly from different frameworks. IBIS is designed to transparently differentiate and schedule application I/Os on every datanode by interposing upon the distributed file system commonly used in big-data systems. It includes a new proportional-share I/O scheduler that can dynamically adjust I/O concurrency to optimize the tradeoff between application fairness and resource utilization. It provides efficient coordination for the I/O schedulers distributed across the datanodes to cooperate and achieve proportional sharing of the big-data system's total I/O service.

The evaluation results confirm that IBIS can effectively achieve total-service proportional bandwidth sharing for diverse applications in the system. They also show that IBIS

can support various important performance polices. It achieves strong *performance isolation* for a less I/O-intensive workload (WordCount, Facebook2009, TPC-H) when under heavy contention from a highly I/O-intensive application (TeraGen and TeraSort), which outperforms native Hadoop by 99% for WordCount and 15% for TPC-H queries. This result is accomplished while still allowing the competing application to make good progress and to fully utilize the storage bandwidth ($<$ 4% reduction in total throughput). IBIS can also achieve excellent *proportional slowdown* for competing applications (TeraSort vs. TeraGen) and outperforms native Hadoop by 30%. Finally, the use of IBIS introduces small overhead in terms of both application runtime and resource usages.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Data-intensive computing systems have been created and evolving to keep up with the pace of the growing needs of computation and data in the modern world. The ubiquitous service implemented by these systems meets the problem of application performance interference in a shared, competing setting adopted by most public service providers or private institutions. These data-intensive applications create bottlenecks on the storage subsystems of the data-intensive computing systems and the bottlenecks have become a major contributor to application performance degradation. In order to service the applications with performance guarantees and efficient resource utilization, this dissertation provides novel approaches to data-intensive storage systems management, including both HPC systems and big-data systems. Specifically, it makes the following three major contributions:

First, virtualization based I/O interposition is introduced upon the storage subsystems of HPC and big-data computing systems. The virtualization technique avoids the intrusive need to change the system internals or the existing application code base. This transparent layer provides the ability to recognize I/Os from different application for further QoS differentiation. In both parallel storage systems and big-data storage systems, the file system layer is chosen to be interposed upon, which is proven be most appropriate as it provides enough higher level semantics and lower level controls.

Second, novel SFQ(D) based proportinoal share algorithms are created on the above-mentioned virtualization layers for HPC and big-data storage systems. On on hand, parallel storage systems face the prominent need of servicing small, latency-sensitive I/Os together with large, sequential I/Os that tend to overload the storage and sabotage the

original SFQ(D) algorithm's fairness in I/O bandwidth sharing. On the other hand, distributed storage systems receive diverse I/Os from different phases with changing I/O concurrency and they saturate the disks frequently. This also undermines the ability of original SFQ(D) algorithm to enforce I/O bandwidth fairness by increasing the queueing time tremendously. To address these issues, the proposed SFQ(D)+ employs a mechanism to differentiate the cost of I/Os of different sizes when being executed in the storage, while SFQ(D2) adapts the I/O concurrency of the storage, both improving the fairness in I/O bandwidth sharing.

Third, for total service proportional sharing, both types of storage systems require coordinated scheduling among the distributed storage nodes. Because of the vast difference in these two systems' architectures, two different approaches are created in order to achieve the same goal. In parallel storage systems, the two proposed synchronization schemes can complement each other by either balancing the cost of synchronization and the unfairness bound or eliminating all-to-all communication when certain I/O distribution information is available. In big-data storage systems, global I/O distrubtion for each application is collected and disseminated through a centralized information broker in a scalable manner.

Specifically, vPFS is created to address the parallel storage management in HPC systems. Today's parallel storage systems are unable to recognize applications' different I/O workloads and to satisfy their different I/O performance requirements. vPFS addresses this problem through the virtualization of contemporary parallel file systems. Such virtualization allows virtual parallel file systems to be dynamically created upon shared physical storage resources on a per-application basis, where each one gets a specific share of the overall I/O bandwidth. This virtualization layer is implemented via parallel file system proxies which interpose between native clients and servers and capture and forward the native requests according to the scheduling policies.

Upon the vPFS framework, a new proportional sharing I/O scheduler, SFQ(D)+, is proposed to allow applications with diverse I/O sizes and issue rates to share the storage with good application-level fairness and system-level utilization. SFQ(D)+ improves over SFQ(D) by taking into account the different I/O depth costs when dispatching small vs. large I/Os, thereby achieving better fairness among diverse workloads. vPFS also includes a combination of efficient synchronization schemes for a large number of distributed schedulers to coordinate their local I/O scheduling and achieve the global, total-service sharing target. These enhancements address the challenges of applying the original SFQ algorithms [JCK04] [WM07] to HPC parallel storage and enhance them for providing high throughput and fair data and metadata services to diverse HPC applications.

The dissertation presents a comprehensive evaluation of the vPFS prototype implemented by virtualizing PVFS2. The results obtained using typical HPC benchmarks and real-world applications, IOR [SS07] and BTIO [BBB+91], WRF [WB05], multi-md-test[CLRT00] show that the vPFS approach is feasible because of its small overhead in terms of throughput and resource usage ($<3\%$ for reads, $<3\%$ for writes, and $<3\%$ for metadata accesses). Meanwhile, it achieves nearly perfect total-service proportional bandwidth sharing for competing parallel applications with diverse I/O patterns ($>96\%$ of the target sharing ratio). vPFS achieves 8.25 times better performance isolation than the native PVFS2, and its SFQ(D)+ scheduler achieves 3.35 times better performance isolation than the original SFQ(D). It also makes efficient use of the storage and the SFQ(D)+ scheduler achieves 13.81 times better total throughput than a non-work-conserving scheduler.

Specifically for big-data systems, IBIS, an Interposed Big-data I/O Scheduler, is created to address the above challenges and provide the much needed I/O performance differentiation to diverse big-data applications, possibly from different frameworks. IBIS

is designed to transparently differentiate and schedule application I/Os on every datanode by interposing upon the distributed file system commonly used in big-data systems. It includes a new proportional-share I/O scheduler that can dynamically adjust I/O concurrency to optimize the tradeoff between application fairness and resource utilization. It provides efficient coordination for the I/O schedulers distributed across the datanodes to cooperate and achieve proportional sharing of the big-data system's total I/O service. The results from an extensive evaluation confirm that IBIS can effectively address the severe I/O interference problem that existing big-data systems have and provide strong performance isolation with efficient resource usage.

The IBIS prototype is implemented in Hadoop/YARN, a widely used big-data system, by interposing HDFS as well as the related local and network I/Os transparently to the applications, and it is able to support the I/O management of diverse applications from different big-data frameworks. It is evaluated using a variety of representative big-data applications (WordCount, TeraSort, TeraGen, Facebook2009 [SWI], TPC-H queries on Hive [TSJ+10]). The results confirm that IBIS can effectively achieve total-service proportional bandwidth sharing for diverse applications in the system. They also show that IBIS can support various important performance polices. It achieves strong *performance isolation* for a less I/O-intensive workload (WordCount, Facebook2009, TPC-H) when under heavy contention from a highly I/O-intensive application (TeraGen and TeraSort), which is better than native Hadoop by 99% better for WordCount and 15% better for TPC-H queries. This result is accomplished while still allowing the competing application to make good progress and to fully utilize the storage bandwidth ($< 4\%$ reduction in total throughput). IBIS can also achieve perfect *proportional slowdown* for competing applications (TeraSort vs. TeraGen) and outperforms native Hadoop by 30%. Finally, the use of IBIS introduces small overhead in terms of both application runtime and resource usages.

5.2 Future Work

This dissertation focuses on the I/O performance guarantee on the storage side in data-intensive computing systems. It immediately spawns further promising research topics to the management of application performance in data-intensive computing systems. As achieving user-perceived application performance is the ultimate goal in large scale, shared data-intensive computing systems, the next three future topics cover different I/O scheduling policies, integration with other resource management, and hollistic end-to-end control of application performance respectively.

5.2.1 Latency-driven I/O Scheduling

The future work will consider other storage management objectives for data-intensive computing systems, and extend it beyond proportional bandwidth sharing. While the differentiation of applications and their respective throughput needs are addressed in this dissertation, a limitation of the proposed storage management approaches is the inability to satisfy both I/O bandwidth and latency needs — current work can only guarantee proportional sharing of bandwidth for the entire system with good utilization.

Latency of an application is the major performance concern for highly interactive applications emerging in the use of HPC in recent years. For example, compared with data mining or hurricane forecast applications which process data in bulks and need a stable, constant throughput regardless of I/O response time, an interactive application such as visualization software may need relatively low throughput but it is sensitive to I/O delays and requires bounded response time. In big-data systems, small jobs also make up the majority of workload composition in production environments (Facebook) and they need bounded I/O latency. Existing related work addresses latency and throughput control in a single-disk system [ZSW$^+$06], but does not provide synchronized latency provisioning

from multiple involved storage nodes, while our initial results on a two-level I/O scheduler that supports both throughput- and latency-driven parallel I/O scheduling on vPFS has shown promising results [XZ13].

Our preliminary two-level scheduler consists of an upper level responsible for request admission and throughput guarantee. It divides the I/O requests into groups and uses SFQ(D) algorithm to throttle and guarantee I/O throughput for each application. The SFQ(D) serves as a credit scheduler, using D as credits for each application bucket. D is measured in bytes instead of storage concurrency slots, replenished globally for all applications, and is thus work-conserving for each application. The lower level consists of an EDF scheduler maintaining the deadline of I/Os dispatched to the storage device. By observing the storage latency, the lower level reports storage idleness to the upper level. It also uses the latency to determine the storage concurrency, improving utilization. This process is repeated online using a feedback-control loop and each action/observation happens within their respective time window.

The first version of this scheduler has been proven effective on a single-node setup of HPC storage systems. The feedback loop can react to system saturation using two distinctive workloads with different throughput and latency QoS demands. It also reacts to deadline misses and storage idleness, maintaining the storage at a balance point where all deadlines are met and throughput is high. On-going work is focused on providing total-service throughput guarantee from the upper admission level, and synchronized application-level latency guarantee on the lower level across many storage nodes. Such a scheduler is also conceived to be feasible on big-data systems which require low delay in job execution, especially small jobs.

5.2.2 Coordinated Storage and Network I/O Management

In our proposed IBIS for big-data storage management, the network throughput guarantee is provided by indirectly controlling the I/O service before receiving/transmitting the network packets. For network reads, the corresponding I/O service is throttled by the local disk reads; for network writes the corresponding I/O service is throttled by the local disk writes. While it proves effective in our setup, it still remains to be solved if the network contention already exists in the link. Initial results have shown that various data-intensive applications can saturate the network link frequently. It is caused by either a large shuffling phase in MapReduce involving an all-to-all traffic pattern, or by an intensive write phase (e.g. in reduce phase), where the replication factor is not low, amplifying the network load. Previous works have either only dealt with network as a single resource not serving storage purposes, or not in coordination with storage I/O management. This lack of coordination of storage and network I/O management impacts both HPC storage systems and big-data storage systems.

For big-data storage systems, complex workload models and patterns exercise the network in multiple ways. Persistent I/Os access the network resource when HDFS I/Os are remote to the task, e.g. a remote HDFS read from a task where the data is not resident on the same node, or when an HDFS write of the final results where the replication factor is larger than two. The majority of intermediate I/Os are also transferred over the network to merge the map results for reducing. This constitutes the main usage of network resource in MapReduce-type big-data workloads. Furthermore, the I/O performance of the data transferred on the network also affects the synchronization traffic. When synchronization messages cannot arrive on time, the service fairness can fluctuate and diverge from the target. All these can result in the cascading of loss of QoS onto the next phase of applications' execution.

In HPC storage systems, network traffic is also outstanding. HPC systems often have high compute-to-storage node ratio, consolidating more data onto smaller number of storage nodes. Thus a higher rate of network transfer is generated on the links, and it may become a bottleneck in the application performance. Currently the network control is not implemented inside HPC storage systems, and a lack of network control method could result in suboptimal performance isolation for some highly latency-sensitive applciations running in HPC systems.

With the advent of Software Defined Network (SDN), the network contention can be managed by applying scheduling algorithms to the switch layer. More importantly, this scheduling must coordinate with the storage management layer on top of the data-intensive computing system. The initial research has rudimentary control over network bandwidth using Open vSwitch and OpenFlow. Challenge remains to dynamically track the network flow and identify which flow belongs to which application, meanwhile maintaining a high overall network utilization.

5.2.3 End-to-end Application Performance Guarantee

The virtualization framework of the data-intensive computing systems enables a variety of interesting future research topics. In HPC systems, the compute nodes are the control knob for the CPU resource, and network controller is the control knob for network resource, and the vPFS is the control knob for its storage system. In big-data systems, the control knob for CPU is number of map slots from the job scheduler, the memory is controlled by Java Virtual Machine launch parameter, the network controller is the knob for network resource, and I/O bandwidth is controlled by IBIS. With all those control knobs, the next step is to automatically and precisely tune these control knobs online holistically. Autonomic resource management will be studied to accurately allocate different

types of resources based on performance models that can capture a MapReduce application's resource demands. For example, different resources' usages will be modeled and the coordination of all resource managers will be designed and implemented.

To deliver guaranteed performance for each application, the mapping between the provisioning of resources and the resulting performance is need, which is called a resource-performance model. To be specific, the model is a mapping from a vector of resources to application performance metrics, such as (CPU, memory, network, storage)$\rightarrow$(throughput / latency / runtime). When equipped with per-application resource-performance models, the resource management system solves an optimization problem within a total resource limit constraint, and a data locality constraint, to maximize application performance and minimize total resource usage. Techniques such as Dominant Resource Fairness can be used on multiple types of resources, while modeling of resource-performance of applications can be done by online fuzzy modeling techniques [WXZF11]. The fuzzy modeling technique has been used on the VM resource allocation in the cloud computing systems, but not yet on the parallel applications in data-intensive systems [WXZ12]. The strength of this tool is that it can characterize highly complex and nonlinear systems, but the integration of network and storage I/O has not been explored either. Although it promises online adaptive training but its effectiveness in diverse workloads and frequent application phase changes also needs to be investigated.

[AEHH+11] Sadaf R. Alam, Hussein N. El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel I/O and the metadata wall. In *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW '11, pages 13–18, New York, NY, USA, 2011. ACM.

[AGW+12] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[AZ11] Dulcardo Arteaga and Ming Zhao. Towards scalable application checkpointing with parallel file system delegation. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 130–139. IEEE, 2011.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks – summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[BBG+99] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems - Volume 2*, ICMCS '99, pages 400–, Washington, DC, USA, 1999. IEEE Computer Society.

[BCG+10] John Bent, HB Chen, David Gunter, Gary Grider, Sam Gutierrez, Adam Manzanares, Ben McClelland, Dave Montoya, James Nunez, Alfred Torrez, Meghan Wingate, Garth Gibson, Milo Polte, and Paul Nowoczinski. PLFS update. High End Computing and File System I/O Workshop, 2010.

[BGK+12] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. Storage challenges at los alamos national lab. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5, 2012.

[CAP+03] D.D. Chambliss, G.A. Alvarez, P. Pandey, D. Jadav, Jian Xu, R. Menon, and T.P. Lee. Performance virtualization for large-scale storage systems. In

Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on, pages 109 – 118, oct. 2003.

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[CLR+09] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[CLRT00] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, ALS'00, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[con] Linux containers. `https://linuxcontainers.org`.

[CST+11] Yong Chen, Xian-He Sun, Rajeev Thakur, Philip C. Roth, and William D. Gropp. LACIO: A new collective I/O strategy for parallel I/O systems. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 794–804, Washington, DC, USA, 2011. IEEE Computer Society.

[DCF03] Aaron E. Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiBLAST. In *In Proceedings of ClusterWorld 2003*, 2003.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[Din09] Tien Duc Dinh. Tracing internal behavior in pvfs. In *Bachelor Thesis*. Ruprecht-Karls-University of Heidelberg, 2009.

[fai] Hadoop Fair Scheduler. `https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`.

[GAW09] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *Proceedings of 7th USENIX Conference on File and Storage Technologies*, 2009.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM Symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[GMV10] Ajay Gulati, Arif Merchant, and Peter J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[GRZ13] Yanfei Guo, Jia Rao, and Xiaobo Zhou. iShuffle: Improving Hadoop performance with shuffle-on-write. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*, pages 107–117, San Jose, CA, 2013. USENIX.

[GV05] Ajay Gulati and Peter Varman. Lexicographic QoS scheduling for parallel I/O. In *Proceedings of the seventeenth Annual ACM Symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 29–38, New York, NY, USA, 2005. ACM.

[had] Apache Hadoop. http://hadoop.apache.org/.

[Hba] HBase. http://hbase.apache.org.

[HH05] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '05, pages 18–27, Washington, DC, USA, 2005. IEEE Computer Society.

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[HPC04] Lan Huang, Gang Peng, and Tzi-cker Chiueh. Multi-dimensional storage virtualization. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 14–24, New York, NY, USA, 2004. ACM.

[inf] InfiniBand. http://www.infinibandta.org/.

[IRYB08] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 153–162, New York, NY, USA, 2008. ACM.

[JCH84] K. Rajandra Jain, W. Dah-Ming Chiu, and R. William Hawe. A quantitative measure of fairness and discrimination of resource allocation in shared computer system. In *DEC Research Report TR-301*, 66 Reed Road, Hudson, MA 01749, USA, 1984. Eastern Research Lab, Digital Equipment Corporation.

[JCK04] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of ACM SIGMETRICS*, pages 37–48, 2004.

[JN99] James Patton Jones and Bill Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '99/JSSPP '99, pages 1–16, London, UK, UK, 1999. Springer-Verlag.

[JSPW+11] Stephanie N. Jones, Christina R. Strong, Aleatha Parker-Wood, Alexandra Holloway, and Darrell D. E. Long. Easing the burdens of HPC file management. In *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW '11, pages 25–30, New York, NY, USA, 2011. ACM.

[KKZ05] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, November 2005.

[Klo10] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[KMT98] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. In *Journal of the Operational Research society*, pages 237–252, London, United Kingdom, 1998. Macmillan Press.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LMA03] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *FAST*. USENIX, 2003.

[LRT04] Rob Latham, Rob Ross, and Rajeev Thakur. The impact of file systems on MPI-IO scalability. In *In Proceedings of EuroPVM/MPI 2004*, 2004.

[Lus08] Lustre file system: High-performance storage architecture and scalable cluster file system. Sun Microsystems White Paper, October 2008.

[LWG+15] 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*, HPDC '15, pages 33–44, New York, NY, USA, 2015. ACM.

[MBL+11] Michael Moore, David Bonnie, Walt Ligon, Nicholas Mills, Shuangyang Yang, Becky Ligon, Mike Marshall, Elaine Quarles, Sam Sampson, and Boyd Wilson. OrangeFS : Advancing PVFS. In *Work-in-progress of the 9th*

USENIX Conferene on File and Storage Technologies, FAST '11, Berkeley, CA, USA, 2011. USENIX Association.

[MBO+11] V. Meshram, X. Besseron, Xiangyong Ouyang, R. Rajachandrasekar, R.P. Darbha, and D.K. Panda. Can a decentralized metadata service layer benefit parallel filesystems? In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 484–493, Sept 2011.

[MW00] Jeonghoon Mo and Jean Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Trans. Netw.*, 8(5):556–567, October 2000.

[NLC08] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 9:1–9:12, Piscataway, NJ, USA, 2008. IEEE Press.

[OAT+07] Ron A Oldfield, Sarala Arunagiri, Patricia J Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C Roth. Modeling the impact of checkpoints on next-generation systems. In *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, pages 30–46. IEEE, 2007.

[OHS+05] Li Ou, Xubin He, S.L. Scott, Zhiyong Xu, and Yung-Chin Fang. Design and evaluation of a high performance parallel file system. In *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, pages 100–107, Nov 2005.

[ope] OpenFlow. `https://www.opennetworking.org/sdn-resources/openflow`.

[PD04] Fabrizio Petrini and Kei Davis. Tutorial: Achieving usability and efficiency in large-scale parallel computing systems. Euro-Par, 2004.

[PG11] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[PSB10] Anna Povzner, Darren Sawyer, and Scott Brandt. Horizon: Efficient deadline-driven disk I/O management for distributed storage systems. In

Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10), pages 1–12, New York, NY, USA, 2010. ACM.

[RD90] R. Rew and G. Davis. NetCDF: An interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, July 1990.

[RI01] Robert B. Ross and Walter B. Ligon III. Server-side scheduling in cluster parallel I/O systems. *Calculateurs Parallles Journal*, November 2001.

[SCZL96] Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. The EASY - LoadLeveler API project. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96, pages 41–47, London, UK, UK, 1996. Springer-Verlag.

[SEP+97] Steven R. Soltis, Grant M. Erickson, Kenneth W. Preslan, Matthew T. O'keefe, and Thomas M. Ruwart. The global file system: A file system for shared disk storage, 1997.

[SFS12] David Shue, Michael J Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 349–362, 2012.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2002.

[SHC+06] Ramanan Sankaran, Evatt R Hawkes, Jacqueline H Chen, Tianfeng Lu, and Chung K Law. Direct numerical simulations of turbulent lean premixed combustion. *Journal of Physics: Conference Series*, 46(1):38, 2006.

[SKRC10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[SLG+09] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic resource allocation for database servers running on virtual storage. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 71–84, Berkeley, CA, USA, 2009. USENIX Association.

[SMMB08] Zoe Sebepou, Kostas Magoutis, Manolis Marazakis, and Angelos Bilas. A comparative experimental study of parallel file systems for large-scale data processing. In *First USENIX Workshop on Large-Scale Computing*, LASCO'08, pages 5:1–5:10, Berkeley, CA, USA, 2008. USENIX Association.

[SS07] H. Shan and J. Shalf. Using IOR to analyze the I/O performance for HPC platforms. In *Cray Users Group Meeting (CUG)*, pages 7–10, 2007.

[Sta06] Garrick Staples. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[SV98] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for next generation operating systems. Technical report, Austin, TX, USA, 1998.

[SWI] Statistical workload injector for mapreduce (SWIM). `https://github.com/SWIMProjectUCB/SWIM/wiki`.

[SYS+11] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. Server-side I/O coordination for parallel file systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 17:1–17:11, New York, NY, USA, 2011. ACM.

[TGL96] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers '96., Sixth Symposium on the*, pages 180–187, 1996.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–, Washington, DC, USA, 1999. IEEE Computer Society.

[tpc] TPC-H Benchmark Specification. `http://www.tpc.org/tpch`.

[TSJ+10] A. Thusoo, J.S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - A petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE'10)*, pages 996–1005, March 2010.

[VMD+13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, and Siddharth Seth. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the Fourth ACM Symposium on Cloud Computing*, 2013.

[Vol] Project Voldemort. `http://www.project-voldemort.com/voldemort/`.

[WB05] Patrick T. Welsh and Peter Bogenschutz. Weather research and forecast model: Precipitation prognostics from the WRF model during recent tropical cyclones, 2005.

[WdW03] Parkson Wong and Rob F. Van der Wijngaart. NAS parallel benchmarks I/O version 2.4. In *NAS Technical Report NAS-03-002*, Moffett Field, CA 94035-1000, USA, 2003. Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center.

[WM07] Yin Wang and Arif Merchant. Proportional-share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[WPBM04] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 4–, Washington, DC, USA, 2004. IEEE Computer Society.

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

[WVA+12a] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC'12)*, pages 14:1–14:14, New York, NY, USA, 2012. ACM.

[WVA+12b] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Ion Stoica, and Randy Katz. Sweet storage SLOs with Frosting. In *Proceedings of the 4th*

USENIX Conference on Hot Topics in Cloud Ccomputing, HotCloud'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[WXZ12] L. Wang, J. Xu, and M. Zhao. Modeling VM performance interference with fuzzy mimo model. In *Proceedings of the 7th International Workshop on Feedback Computing*, 2012.

[WXZF11] Lixi Wang, Jing Xu, Ming Zhao, and Jose Fortes. Adaptive virtual resource management with fuzzy model predictive control. In *Proceedings of 6th International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID, co-held with ICAC)*, June 2011.

[XAZ+12] Y. Xu, D. Arteaga, M. Zhao, Y. Liu, R. Figueiredo, and S. Seelam. vPFS: Virtualization-based bandwidth management for parallel storage systems. In *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST)*, April 2012.

[XZ13] Yiqi Xu and Ming Zhao. Two-level throughput and latency I/O control for parallel file systems. In *Proceedings of the 8th International Workshop on Feedback Computing (co-held with ICAC)*, 2013.

[YLP05] Weikuan Yu, Shuang Liang, and Dhabaleswar K. Panda. High performance support of parallel virtual file system (PVFS2) over Quadrics. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 323–331, New York, NY, USA, 2005. ACM.

[YSH+06] H. Yu, R.K. Sahoo, C. Howson, G. Almasi, J.G. Castanos, M. Gupta, J.E. Moreira, J.J. Parker, T.E. Engelsiepen, R.B. Ross, R. Thakur, R. Latham, and W.D. Gropp. High performance file I/O for the Blue Gene/L supercomputer. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 187 – 196, feb. 2006.

[ZDJ10] Xuechen Zhang, Kei Davis, and Song Jiang. IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[ZDJ11] Xuechen Zhang, Kei Davis, and Song Jiang. QoS support for end users of I/O-intensive applications using shared storage systems. In *Proceedings of 2011 International Conference for High Performance Computing, Net-*

working, Storage and Analysis, SC '11, pages 18:1–18:12, New York, NY, USA, 2011. ACM.

[ZF06] Ming Zhao and R. J. Figueiredo. Application-tailored cache consistency for wide-area file systems. In *Proceedings of Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 41–41, 2006.

[ZSW+06] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. *ACM Transactions on Storage*, 2(3):283–308, August 2006.

[ZZF06] Ming Zhao, Jian Zhang, and Renato J. Figueiredo. Distributed file system virtualization techniques supporting on-demand virtual machine environments for grid computing. *Cluster Computing*, 9(1):45–56, January 2006.

VITA

YIQI XU

	Born, Shanghai, P.R.China
2004	B.S., Computer Science Fudan University Shanghai, P.R.China
2004–2009	Senior Analyst Dell China Design Center Shanghai, P.R.China
2012	M.S., Computer Science Florida International University Miami, Florida
2014	VMware Academic Program Graduate Fellowship Award
2009-2016	Doctoral Candidate in Computer Science Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Yiqi Xu, Ming Zhao, "IBIS: Interposed Big-data I/O Scheduler," *In Proceedings of 25th International ACM Symposium on High Performance Parallel and Distributed Computing* (HPDC 2016), Kyoto, Japan, June 2016

Yiqi Xu, Saman Biookaghazadeh, Shujia Zhou, Ming Zhao, "Enabling Scientific Data Storage and Processing on Big-data Systems," Big Data (Big Data), *Geosciences Workshop in 2015 IEEE International Conference on*, Santa Clara, CA, Oct 2015

Michel Roger, Yiqi Xu, Ming Zhao, "BigCache for Big-data Systems," Big Data (Big Data), *2014 IEEE International Conference on*, Washington DC, Oct 2014

Yiqi Xu, Adrian Suarez, Ming Zhao, "Interposed Big-data I/O Scheduler," Short Paper *In Proceedings of 22nd International ACM Symposium on High Performance Parallel and*

Distributed Conference (HPDC 2013), New York City, NY, June 2013

Yiqi Xu, Ming Zhao, "Two-level Throughput and Latency IO Control for Parallel File Systems," *In Proceedings of 8th International Workshop on Feedback Computing* (Feedback 2013, in 2013 USENIX Federated Conferences Week), San Jose, CA, June 2013

Yonggang Liu, Renato Figueiredo, Yiqi Xu, Ming Zhao, "On the Design and Implementation of a Simulator for Parallel File System Research," *IEEE 29th Symposium on Massive Storage Systems and Technologies* (MSST 2013), Long Beach, CA, May 2013

Yiqi Xu, Adrian Suarez, Ming Zhao, "IBIS: Interposed Big-data I/O Scheduler," Work in Progress and Poster, *USENIX 11th Conference on File and Storage Technology* (FAST 2013), San Jose, CA, Feb 2013

Yiqi Xu, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato Figueiredo, Seetharami Seelam, "vPFS: Bandwidth Virtualization of Parallel Storage Systems," *In Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies* (MSST 2012), Pacific Grove, CA, April 2012

Yiqi Xu, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato Figueiredo, Seetharami Seelam, "vPFS: Performance Virtualization of Parallel Storage Systems," Poster, *USENIX 10th Conference on File and Storage Technology* (FAST 2012), San Jose, CA, Feb 2012

Yiqi Xu, Lixi Wang, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato Figueiredo, Seelam Seetharami, "vPFS: Bandwidth Virtualization of Parallel Storage Systems," *Technical Report 2010-05-01*, SCIS, Florida International University, 2011

Yonggang Liu, Renato Figueiredo, Dulcardo Arteaga, Yiqi Xu, Ming Zhao, "Towards Simulation of Parallel File System Scheduling Algorithms with PFSsim," *In Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O* (SNAPI 2011, co-held with MSST 2011), Denver, CO, May 2011

Yiqi Xu, Lixi Wang, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato Figueiredo, "Virtualization-based Bandwidth Management for Parallel Storage Systems," *In Proceedings of 5th Petascale Data Storage Workshop* (PDSW 2010, co-held with SC 2010), New Orleans, LA, November 2010