T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Design, Optimization and Evaluation of a Hierarchical Filesystem
著者(和文)	徐天棋
Author(English)	Tianqi Xu
出典(和文)	学位:博士(学術), 学位授与機関:東京工業大学, 報告番号:甲第11025号, 授与年月日:2018年12月31日, 学位の種別:課程博士, 審査員:松岡 聡,増原 英彦,遠藤 敏夫,脇田 建,額田 彰
Citation(English)	Degree:Doctor (Academic), Conferring organization: Tokyo Institute of Technology, Report number:甲第11025号, Conferred date:2018/12/31, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Science Tokyo Research Repository)

Design, Optimization and Evaluation of A Hierarchical Filesystem

by

Tianqi Xu

徐 天棋

A Thesis Submitted to

Dept. of Mathematical and computing sciences

Tokyo institute of technology

In Partial Fulfillment of the Requirement for the Degree of

DOCTOR OF PHILOSOPHY

Adviser: Professor Satoshi Matsuoka

June 2018

Abstract

High performance computing (HPC) centers have been utilized to advance scientific discoveries for decades with their state-of-the-art computational performance. By running in large-scale, the run time of applications that handle big-data can be reduced from days and years to hours. The computational performance has been growing dramatically fast, which help to handle such large-scale applications more efficiently. However, the performance of the storage system can hardly catch up with the performance increment of the computation, which makes the storage system a bottleneck in the large-scale clusters. Moreover, with the increasing amount of data being produced every day to be analyzed, the performance gap between the computational and the storage system exacerbated.

In this thesis, we conduct three works to address such challenges in two major kinds of clusters for running large-scale data-intensive applications, the HPC centers and Clouds. Although by providing high performance instances, such as the HPC instance on Amazon AWS, the Cloud matches the computational performance with the HPC centers, the performance of the storage systems is still far behind the performance of the parallel file system in HPC centers. The major Cloud storage systems are built on object storage technologies, for better load-balancing, scalability and availability. However, the object storage introduces additional limitations to the workload, such as the N-1 write and consistency issues, which limits the performance of applications, especially for the large-scale data-intensive processing. Such limitations make the storage the bottleneck of the system while executing large-scale data-intensive applications, degrades the performance of the applications, and costs users more due to the Cloud pricing policy, To help to improve the performance of the storage on the Cloud, we propose a software-level burst buffer system, called CloudBB. CloudBB utilizes instances as a temporary buffer space to buffer the I/O data from other instances. By buffering the intermediate data within a high-performance network, CloudBB accelerates both the read and write operations from the applications. Moreover, we propose a Master-Worker and Key-Value architecture to achieve both scalability while maintaining the high metadata performance. We implemented CloudBB on top of FUSE so that the applications need no code modification to use CloudBB. We evaluated our CloudBB on Amazon AWS Cloud environment with both micro-benchmarks and real applications. We observed significant performance improvement against the Amazon S3 storage.

Even though the HPC centers equipment with much higher performance parallel file systems compared to the Cloud, with the dramatic increment of the computational performance in HPC, the performance of the parallel file systems can hardly catch up, with multiple users sharing the same file system, the parallel file systems become the bottleneck in the HPC centers for executing large-scale data-intensive applications. Burst buffer systems have been deployed in the latest cutting-edge HPC centers to buffer the bursty I/O and improve the I/O performance. However, having burst buffer systems involves additional procedures including procurement, deployment and maintenance. Therefore, it is not economically nor logistically feasible to have a burst buffer system in every HPC center. Moreover, physically deployed burst buffer has its designed capacity and performance, which can hardly adapt to any demand changes. In order to solve such problems, we extend our CloudBB to the HPC environments with high-performance network supports and propose a software- and user-level on-demand burst buffer system, HuronFS. Similar to the CloudBB, HuronFS is designed as a software burst buffer utilizing compute nodes with high-performance network to buffer the I/O data, hence HuronFS can be easily deployed on any HPC centers. Furthermore, thanks to the Master-Worker and Key-Value architecture, the capacity and the performance of HuronFS can be adapted to the workload. We evaluted our HuronFS on two supercomputer systems and demonstrated that using HuronFS can achieve the performance of state-of-the-art parallel file systems and help users to improve the I/O performance in the HPC centers.

Utilizing the burst buffer systems has been proven to accelerate the I/O performance. To use a burst buffer, the users need to specify the configurations of burst buffer in the job scripts, i.e. buffer size. However, the configurations must be carefully determined to avoid experiencing poor performance or causing the low job throughput due to under-utilization. To understand the performance impacts from different burst buffer configurations, we conduct a performance analysis with a trace-driven simulator. We collect I/O trace from real applications and simulate their performance under different configurations. We explore three different configurations aspects, the swap-in/out granularity; different buffer size; and different data replacement algorithms. The simulation results show that different applications have different requirements for burst buffer configurations and help to further optimize the performance using burst buffer.

Our studies of design optimization and evaluation of burst buffer in both HPC centers and Cloud helps to alleviate the performance gap between the computation

and storage systems, helps to further understand and optimize the burst buffer performance to support the future large-scale data-intensive applications.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor prof. Satoshi Matsuoka for the continuous support of my Master study and research, for his patience, motivation, enthusiasm, and immense knowledge. He provided me the chance to study and research in his lab with lots of outstanding staffs and colleagues. His guidance helped me in all the time of research and writing of this thesis. I am also grateful to the members of my committee for their advice and comments. Secondly, I would like to thank my advisor Dr. Kento Sato for his countless advice and helps. He took a lot of time to discuss with me every week, guided me in my research, programming, system design and writing. I would like to thank for all staffs and colleagues in matsuoka lab, thank you for your helps in my research and this thesis. Finally, I would like to thank for my parents for their supports and helps in this thesis and my life in general.

Contents

Abstract . . . ii
Acknowledgements . . . v
List of Tables . . . ix
List of Figures . . . x

1 Introduction **1**
1.1 Utilizing Cloud for Large-scale Data-intensive Applications . . . 1
1.2 Software-Level Burst Buffer System for HPC . . . 3
1.3 Analysis of the Configurations of Burst Buffer for High Performance . 4
1.4 Our Contributions . . . 5
1.5 Structure of This Thesis . . . 7

2 Background and Motivation **8**
2.1 Data Intensive Applications . . . 8
2.1.1 Introduction of Data Intensive Applications . . . 8
2.1.2 Temporal I/O Locality in Data Intensive Workloads . . . 9
2.2 Overview of Cloud Computing . . . 11
2.3 Burst Buffer . . . 15
2.3.1 Overview of Burst Buffer . . . 15
2.4 Storage System in Cloud and HPC . . . 17
2.4.1 HPC Storage System . . . 17
2.4.2 Cloud Storage System . . . 25
2.4.3 Problems in Cloud Storage Systems . . . 29
2.5 FUSE . . . 32
2.6 CCI . . . 36
2.7 Key Configurations in RSBB . . . 37
2.7.1 Allocation Size . . . 37
2.7.2 Data Replacement Algorithm . . . 38
2.7.3 Summary . . . 38

2.8 The Configurations of Burst Buffer 38

3 Utilizing Cloud for Large-scale Data-intensive Applications 40

3.1 CloudBB Architecture and Implementation 40

3.1.1 Scalable Multi-Master-Worker Architecture 41

3.1.2 Hierarchical I/O Operations 42

3.1.3 Fault Tolerant Filesystem 43

3.1.4 Highly Portable Filesystem Implementation 45

3.2 Implementation 45

3.2.1 Implementation Overview 46

3.2.2 Optimizations for Performance 46

3.2.3 Architecture and Internal Data Structure 49

3.2.4 Master 49

3.2.5 IOnode 53

3.2.6 Client 56

3.2.7 File Operations 61

3.3 Optimizations 67

3.3.1 Socket Reuse 68

3.3.2 File Caching 68

3.3.3 Metadata Caching 69

3.4 Evaluation 70

3.4.1 Comprehensive I/O Pattern Evaluations 71

3.4.2 Failure Recovery 73

3.4.3 Case Studies in Real Applications 74

4 Software-Level Burst Buffer System for HPC 83

4.1 Motivation of Having HuronFS 83

4.2 Improvement from CloudBB 84

4.2.1 TCP/IP →CCI 84

4.2.2 Multi-threading Support 85

4.2.3 Limited Buffer Support 85

4.3 Evaluation 86

4.3.1 Basic I/O Performance 87

4.3.2 metadata performance 90

4.3.3 Real Applications 91

5 Analysis of the Configuration of Burst Buffer **96**
5.1 Design In Simulation 96
5.1.1 Trace-driven Simulation 96
5.1.2 FUSE-based I/O Tracing 97
5.2 Burst Buffer Model and Simulator 99
5.3 Simulation Methodology 102
5.3.1 Performance Counters 103
5.3.2 Two Writeback Strategies 105
5.3.3 System Benchmark 106
5.3.4 Distributed I/O Trace 108
5.4 Simulation Results 109
5.4.1 Simulating with Real Applications 110
5.4.2 Chunk-level VS. File-level 124
5.4.3 Simulation with different data replacement algorithms 124
5.4.4 Discussion 129
5.5 Simulation with Parallel Execution 130
5.5.1 Execution Patterns Changed by Parallelism 131
5.5.2 Hypotheses 133
5.5.3 Empirical Studies 140
5.5.4 Reuse Distance 144

6 Related Work **152**

7 Conclusion and Future Work **160**

Reference **162**

List of Tables

2.1 Montage Execution detail 10
2.2 Lustre Performance [13] 22
2.3 Hooks in FUSE 2.9.4 [36] 35

3.1 Master Functions 50
3.2 IOnode Functions 54
3.3 Evaluation environment 71
3.4 Input Data Size and Total IO size of Applications 73
3.5 I/O size of Montage and Supernovae 74

4.1 TSUBAME 2.5 Specifications 86
4.2 TSUBAME KFC Specifications 86
4.3 HuronFS Configurations 86
4.4 Basic I/O Benchmark Details 86
4.5 Real Application Details 92
4.6 Real Application Details 93

5.1 Experiment Environment 109
5.2 Input Data Size and Total access size of Applications 110
5.3 I/O patterns of Applications 110
5.4 HuronFS Configurations for Application 140
5.5 System Specification 140

6.1 Explanation of the decision alghrithm's variables [42] 158

List of Figures

2.1 The Process of Montage [14] . . . 10
2.2 Three Models in Cloud Computing . . . 13
2.3 Public/Private/Hybrid Cloud . . . 14
2.4 Burst Buffer Tier in HPC System [54] . . . 16
2.5 Modular HPC Architecture . . . 17
2.6 Direct Attach SAS Storage . . . 19
2.7 SAN Storage . . . 20
2.8 Parallel I/O Subsystem . . . 20
2.9 Lustre Architecture . . . 21
2.10 PVFS Architecture . . . 22
2.11 An example of File Block List in PVFS . . . 23
2.12 PVFS File Striping Example . . . 23
2.13 GPFS Architecture . . . 24
2.14 Cloud Storage Architecture . . . 25
2.15 Read/write I/O throughput with a single shared file on Amazon s3. . . 29
2.16 Read/write I/O throughput with each nodes accessing its own file on Amazon s3. . . . 30
2.17 Point-to-point communication throughput in Amazon EC2 . . . 31
2.18 FUSE Architecture . . . 33
2.19 Overview of CCI . . . 36
2.20 An example of burst buffers in an HPC system . . . 38
2.21 The I/O performance under different sizes of burst buffers . . . 39

3.1 Architecture of CloudBB . . . 40
3.2 File Chunking and Buffers in Compute Nodes . . . 47
3.3 Master Architecture . . . 50
3.4 Data Structure in Master . . . 51
3.5 IOnode Architecture . . . 54

3.6 Data Structure in IOnode 55
3.7 Block Size and Valid Data Size 55
3.8 Client Architecture 57
3.9 Client Process Detail 58
3.10 Data Structure in Client 58
3.11 Client Buffer 59
3.12 Dataflow of File Operation 61
3.13 Process of open/create file 62
3.14 Process of Write 63
3.15 Process of Read 64
3.16 Process of Flush 66
3.17 Process of Close 67
3.18 Process of gettattr 68
3.19 Process of readdir 69
3.20 Process of unlink 70
3.25 Cost for execution of the Himeno benchmark with Checkpointing . . 82

4.1 The Bandwidth Comparison between TCP/IP and RDMA 84
4.2 The Latency Comparison between TCP/IP and RDMA 85
4.3 Sequential I/O Performance with Single Client on TSUBAME 2.5 . . 87
4.4 Random I/O Performance with Single Client on TSUBAME 2.5 . . . 87
4.5 Sequential I/O Performance with Multiple Clients 88
4.6 Luster Workload on TSUBAME 2.5 89
4.7 Sequential I/O Performance with Multiple Clients on TSUBAME KFC 89
4.8 Random I/O Performance with Multiple Clients on TSUBAME KFC 90
4.9 File Creation Performance with Single Clients on TSUBAME 2.5 . . 90
4.10 File Creation Performance with Multiple Clients on TSUBAME 2.5 . 91
4.11 Lustre Metadata Workload on TSUBAME 2.5 91
4.12 Miranda I/O results on TSUBAME KFC 92
4.13 BTIO results on TSUBAME KFC 93
4.14 Miranda I/O with Limited Buffer on TSUBAME KFC 94
4.15 BTIO with Limited Buffer on TSUBAME KFC 94
4.16 Montage with Limited Buffer on TSUBAME KFC 95

5.1 Burst buffer model 100
5.2 Read in the burst buffer model 100
5.3 Write in the burst buffer model 100

5.4 The overview of our simulation 103
5.5 Examples of Performance Counters 104
5.6 An example of trace files 108
5.7 Performance Counters in Different Colors 111
5.8 Simulation result of Montage 113
5.9 Simulation result of Similar Pages 116
5.10 Simulation result of Povray 118
5.11 Simulation result of BTIO 121
5.12 Simulation result of Miranda_IO 123
5.13 Simulation Result With Chunk-level and File-level swap-in/out 126
5.14 Simulation Result With Different Data Replacement Algorithms . . . 129
5.15 An example of a control flow 131
5.16 An example of a execution sequence 132
5.17 Execution pattern changes due to parallelism 132
5.18 An Example of Parallel Parts with Sequential Parts 135
5.19 The Parallelism Is Not Random Order 135
5.20 An example of reuse distance 136
5.21 Buffer Hit in Both Cases 137
5.22 Buffer Miss in Both Cases 138
5.23 Buffer Miss in One Cases 139
5.24 Simulation Comparison on Montage 141
5.25 Simulation Comparison on BTIO 142
5.26 Simulation Comparison on Miranda 142
5.27 Simulation Comparison on Povray 143
5.28 Simulation Comparison on Similar Pages 143
5.29 An Example of Miscounting 144
5.30 Reuse distance of Montage 145
5.31 Reuse distance of Similar Pages 146
5.32 Reuse distance of Povray 147
5.33 Reuse distance of BTIO 148
5.34 Reuse distance of Miranda_IO 149

6.1 Flat buffer C/R system 152
6.2 Burst buffer C/R system 153
6.3 Architectural overview of XML processing in the cloud to support digital preservation 154

6.4 Single Satellite Ground System 155
6.5 Multiple Satellite Ground Systems 156

Chapter 1

Introduction

Large-scale clusters have been utilized to advance scientific discoveries for decades with their state-of-the-art computational performance. By running in large-scale, the run time of applications that handle big-data can be reduced from days and years to hours. The computational performance has been growing dramatically fast, which help to handle such large-scale applications more efficiently. However, the performance of the storage system can hardly catch up with the performance increment of the computation, which makes the storage system a bottleneck in the large-scale clusters. Moreover, with the increasing amount of data being produced every day to be analyzed, the performance gap between the computational and the storage system exacerbated.

In order to support the future large-scale data-intensive applications, we conduct three works to address three challenges in two major platforms for the large-scale data-intensive applications, the HPC centers and Clouds.

1.1 Utilizing Cloud for Large-scale Data-intensive Applications

Cloud computing has been gathering attentions from application developers in high performance computing (HPC) because of its elasticity. With the elasticity, users can benefit from virtually unlimited computational resources on the fly in a pay-as-you-go model. Furthermore, recent clouds also provide computational resources for HPC [1, 4, 10, 11, 16]. For example, Amazon EC2 provides HPC instances with high bandwidth networks, I/O subsystems with high performance SSDs, and GPUs [22].

These characteristics make cloud computing more attractive for large-scale scientific applications.

However, current public clouds cannot provide enough *performance* and *consistency* for data-intensive HPC applications when using shared cloud storage. In public clouds, each instance is connected to shared cloud storage via Internet. Network throughput between an instance and shared cloud storage becomes performance bottleneck for applications, and the slow I/O performance degrades data-intensive HPC applications running on clouds. For example, Amazon Simple Storage Service (Amazon S3) provides only a few hundreds MB/s of throughput [68], whereas typical parallel file systems (PFS) of HPC systems can serve up to a few TB/s [13].

Another problem is its consistency policy. Shared cloud storage typically replicates stored files for reliability, and maintain consistency among replicas based on *eventual consistency* in favor of improved latency and performance [2, 17, 71]. However, the eventual consistency does not guarantee that one process read the latest version of a file even if another process updates the file ahead of the read request. In practice, multiple processes write, read and update files in data-intensive HPC applications, such applications cannot correctly run on such shared cloud storage [26, 39]. Thus, current shared cloud storage cannot fulfill demands of data-intensive HPC applications.

To solve the performance and the consistency problems, we propose a cloud-based burst buffer (CloudBB). CloudBB is designed to fullfill the demands of data-intensive HPC applications. CloudBB uses instances as dedicated resources for burst buffers. CloudBB locates burst buffers on top of shared cloud storage as additional tier of storage hierarchy. Our analysis shows that most data-intensive HPC applications access to data with high temporal I/O locality. By caching such data on the burst buffers, CloudBB can accelerate I/O performance. In addition, since all data is accessed through CloudBB, consistency can be guaranteed on buffer-level. Furthermore, by combining a Master-Worker model with a Key-Value store model, CloudBB can achieve high scalability in shared storage unlike typical PFS, in which single metadata server usually becomes a bottleneck of the whole system. We implement CloudBB using FUSE [36]. Thus, application developers seamlessly use CloudBB without code modification.

1.2 Software-Level Burst Buffer System for HPC

Secondly, even though the HPC centers provide much faster parallel file systems compared to the Cloud. the growing requirements from the large-scale data-intensive applications, still push the storage systems in HPC centers to their limitation. The computational performance in high performance computing (HPC) systems has been dramatically increased, driven by continuously advancing multi- and many-core architectures and fast memory technologies such as high bandwidth memory (HBM) [46] and hybrid memory cube (HMC) [43]. The fastest supercomputer in the world first reached beyond a hundred PFLOPS in 2016. The advance is a factor of 400 compared to the fastest supercomputer in 2006. Although these recent HPC systems have been keeping pace with the requirements of compute- and memory-intensive applications, the current systems remain inadequate for I/O-intensive applications since the performance of the I/O subsystem, such as a parallel file systems (PFS), is much lower than that of processors and memory systems. As we move towards more high performance and large-scale data-intensive applications on HPC systems, I/O performance has been becoming more critical [85, 87].

To alleviate the gap between computational and I/O performance, production supercomputers have been adopting burst buffer systems [31, 33, 57, 64, 79]. Burst buffers are an additional storage tier, residing on top of the PFS in the storage hierarchy and provide non-volatile storage space with high bandwidth and lower latency than the PFS. Burst buffers are used as either local storage that has been installed on each compute node, *node-local burst buffer* (NLBB) [57, 64, 79] or remote storage that is shared by compute nodes and serve as cache space for PFSs, *remote shared burst buffer* (RSBB) [31, 33, 51, 62, 80].

However, due to the high procurement and maintenance cost of the burst buffer systems, the deployments of burst buffer are very limited to a small group of cutting-edge HPC centers [57, 64, 79, 80], most of the traditional HPC centers do not have the burst buffer system available. In order to help the users in those HPC centers to accelerate their data-intensive applications, we extend our software-level Cloud-based burst buffer in to the HPC environment, and let the user construct a software-level burst buffer system for their running jobs. To fully utilize the high performance networks, in addition to the TCP/IP protocol used in Cloud-based burst buffer, we added the support for high performance networks such as Infiniband by using a communication framework, called CCI. According to the experimental results on

a real HPC system, using HuronFS can significantly improve the performance of applications compare to the NFS storage, and achieve comparable performance of the state-of-the-art parallel file system.

1.3 Analysis of the Configurations of Burst Buffer for High Performance

In order to fully utilize the performance of the burst buffer to help to improve the I/O performance in the HPC centers, finding the right configuration is essential.

In NLBB, burst buffers are allocated to jobs along with the compute nodes so that the scheduled job has dedicated use of the allocated compute nodes' local storage. Therefore, the total size of burst buffers is determined by the number of compute nodes that the job requests. Whereas, in RSBB, the burst buffers are shared by all compute nodes. When running a job, the user specifies the total size of burst buffers to use via APIs [32]. However, while RSBB provides its users with flexibility in specifying the size of burst buffers, the RSBB size must be carefully determined since applications may crash due to I/O errors if applications attempt to use more RSBB capacity than requested size, i.e. overflow. Moreover, intermediate files and different ways of storing file in different filesystems, e.g. Internal fragmentation caused by file chunking, make it difficult to accurately estimate an application's capacity usage, which increases the probability of burst buffer overflow.

The current solution is to allocate a surplus in capacity that exceeds the usage estimate in order to avoid job failure . However, such solution causes significantly under-utilization of burst buffer, reduction in the system's job throughput, and increase in the job queuing time. Another solution is to stage-out data that the application does not use anymore and then stage-in data which the application accesses next, similar to the cache swapping. Recent RSBB systems provide APIs for dynamic stage-in/out functionalities so that the application can process more data than the specified amount [32]. However, the granularity of stage-in/out in recent RSBB systems is *file-level*. This granularity significantly degrades application performance since an entire file must be staged in the RSBB even if the application need to read only a small portion of the file.

We explore the performance impact on I/O-intensive applications when RSBB systems support finer-grained granularity for dynamic stage-in/out: *chunk-level*. This finer-grained granularity raises several interesting questions. Specifically, we

explore the following three critical questions: (i) how different RSBB configurations, i.e., buffer size and bandwidth, impact performance; (ii) how efficiently the chunk-level swap-in/out[1] approach can utilize RSBB; (iii) how different chunk-level data replacement algorithms, i.e., FIFO, LRU and others, impact performance . We investigate these questions with our burst buffer simulator, B2Sim. B2Sim is a trace-driven simulator and the simulation starts with tracing all I/O operations of an application, then B2Sim simulates the execution of the application based on the I/O trace under different configurations, i.e., different buffer sizes, bandwidths and different swap-in/out algorithms.

1.4 Our Contributions

Our contributions can be summarized as followings:

- **A cloud-based burst buffer model as a new tier of storage hierarchy in Clouds;**
  As we mentioned before, we extend burst buffer technology into cloud, use several compute nodes as burst buffer nodes to buffer the I/O data from users' application. We take advantage of high throughput between the compute nodes to accelerate the I/O performance. Our system also solve the eventual consistency issues which causes job failure when using cloud storage.

- **A hybrid Master/Client, Key/Value model for scalable burst buffers;**
  As a burst buffer system in cloud, our system is supposed to be able to add or remove burst buffer nodes on-demand, in order to achieve higher performance or reduce the cost. Furthermore since we buffer I/O data in burst buffer, our system needs to write data back to shared storage and free room for new data to prevent overflow. In order to do achieve these, we need to monitor the whole system. However architectures like Master/Client model usual has problems with scalability, central node can easily become bottleneck of the whole system. In this thesis, we propose a hybrid Master/Client, Key/Value model to gain monitoring of the whole system as well as scalability.

- **An implement of the CloudBB;**
  In order to validate the effectiveness of our proposal system, we implemented

[1] We use *swap-in/out* for moving data *chunks* between burst buffers and a PFS in order to differentiate with traditional stage-in/out of *files*

our proposal system, and we introduced several optimizations for performance. We talk about the details of the implementation in chapter 3.2.

- **Evaluations on the CloudBB with micro benchmarks and real data intensive applications**
  We evaluated CloudBB in a real public cloud, Amazon EC2/S3. We run several micro benchmarks including sequential and random access to show how our system can improve the I/O performance. We also execute real world data intensive applications, Montage and Supernovae, on the top of our system with different number of compute nodes and burst buffer nodes. The results are shown in chapter 3.4.

- **HuronFS (Hierarchical, User-level and ON-demand FileSystem for HPC centers)**
  We extend our CloudBB system to the HPC environments and propose HuronFS (Hierarchical, User-level and ON-demand FileSystem). In addition to the TCP/IP protocol, HuronFS supports the high performance network communication such as Infiniband. By utilizing HuronFS, users in HPC centers without hardware burst buffer system can still enjoy the performance improvements.

- **Evaluations of HuronFS on a real HPC system** We evaluated HuronFS in a real HPC system which does not equipment with physical burst buffer. We run several micro benchmarks and real applications with the HuronFS. We demonstrate that with the help of HuronFS, we can achieve a much high performance than the common NFS and comparable performance of the state-of-art parallel file systems.

- **A Generic Model of Burst Buffer Under A Chunk-level Swap-in/out Scheme**
  We define a generic model of burst buffer. Different from the existing burst buffer model, our model supports execution of applications on a burst buffer system with limited buffer size by utilizing swap-in/out scheme. In addition, our model supports chunk-level swap-in/out for further application support.

- **A simulator That Simulates I/O Behaviors of Applications Based on The Burst Buffer Model**
  We build a simulator, B2sim, base on our burst buffer model to simulate the

performance of applications under different given burst buffer configurations. We build the simulator on top of the trace-driven simulation technology. By simulating with the I/O trace from applications, we provide generality while capturing the details of applications.

- **Comprehensive explorations on a variety of burst buffer configurations**
  We simulate different performance of five different applications under different burst buffer configurations. We provide analysis of three different aspects of configurations.

1.5 Structure of This Thesis

The rest of this paper is organized as follows. In Chapter 2, we describe the background and motivation Then, we describe the design and evaluation of our cloudBB system in Chapter 3. We show how we extend the CloudBB system into HuronFS and the performance evaluation on two HPC systems in Chapter 4. Finally, we analyze the how different configurations impacts on the burst buffer systems in Chapter 5. In Chapter 6 we introduce several related work to our project and conclude this paper in Chapter 7.

Chapter 2

Background and Motivation

In this chapter, we introduce the background information of our project and our motivation. As mentioned in Section 1, our goal is to accelerate data intensive applications on cloud. We give the introduction of data intensive applications in Section 2.1, and cloud in Section 2.2 to explain the goal of this research. Then we introduce the problem statements and our motivation. We present comparison of HPC storage system and cloud storage system in Section 2.4, we show the performance gap between these two storage systems, which is the major challenge for the migration of data intensive applications from HPC to cloud and the consistency issue in common cloud storage systems. Finally, we introduce the technologies we used in our solution in Section 2.3 and Section 2.5.

2.1 Data Intensive Applications

Our approach of accelerating data intensive applications is based on I/O patterns of data intensive applications and characteristics of cloud network environments. In order to explain our approach clearly, we first give brief introduction of data intensive applications (Section 2.1.1), and then explain about the I/O patterns of data intensive applications in Section 2.1.2.

2.1.1 Introduction of Data Intensive Applications

The rapid grows of Internet and science led to eager desire of large data process. Almost all fields have requirement for parsing Big Data especially in science field such as Physic, Astronomy, Chemistry etc.. For example the domain size for stencil computing in weather forecast grows to provide more accurate prediction. Instead of

traditional approach, using parallel processing on multiple process/thread/node can accelerate the applications significantly.

The parallel processing can be classified into *Compute Intensive Applications* and *Data Intensive Applications*. *Compute Intensive Applications* refers to applications which have heavy computing burden. Such applications spend most their time on computation rather than I/O. Compute intensive applications typically involve parallelizing individual algorithms, which split the whole task into several subtasks, and assign each process to handle the subtasks separately.

On the other hand, the term, *Data Intensive Applications*, refers to parallel computing applications which process on large data set or produce large outputs. The volumes of data are typically terabytes or petabytes in size and referred as big data. Such applications spend major their execution time on I/O operations, so the I/O performance of storage has huge impact on their performance. Unlike compute intensive applications, data intensive applications split the whole data set into several sub data sets and assign each process to process some of these sub data sets. Data intensive applications typically run on multiple nodes and issue I/O requests from these compute nodes simultaneously, such characteristic require the storage systems to handle the request parallel. In this paper we focus on data intensive applications.

The key point to accelerate data intensive applications is to build faster and scalable shared storages. Many researches have been devoted on this aspect and many storage systems have been proposed, such as Lustre, Gfarm, HDFS These file systems are widely used in HPC centers and supports the execution of data intensive applications.

2.1.2 Temporal I/O Locality in Data Intensive Workloads

Our analysis of data intensive workloads on HPC systems shows that a number of applications have temporal I/O locality. Figure 2.1 shows the process of a real scientific workflow application, Montage – a portable software toolkit for constructing custom, science-grade mosaics by composing multiple astronomical images [14]. We can see that the whole process can be divided into several sub processes; each sub-process reads the output of previous sub-processes and generates data for successive sub-processes. We found that such a process pattern increases the temporal I/O locality of workflow applications.

	data set 0	data set 1	data set 2
Input data set size (MB)	25	182	1200
Output data set size (MB)	76.5	574.3	7100
Total I/O size (MB)	224.1	2132.4	30250.018
Total read size(MB)	147.6	1558.1	22667.261
Total write size(MB)	76.5	574.3	7582.757049
Temporal I/O locality size(MB)	139.14	2074.5	29609.493
R_{buf}	62%	97.3%	97.882%
Single Thread Compute time (s)	4.7	18.7	311.82

Table 2.1: Montage Execution detail

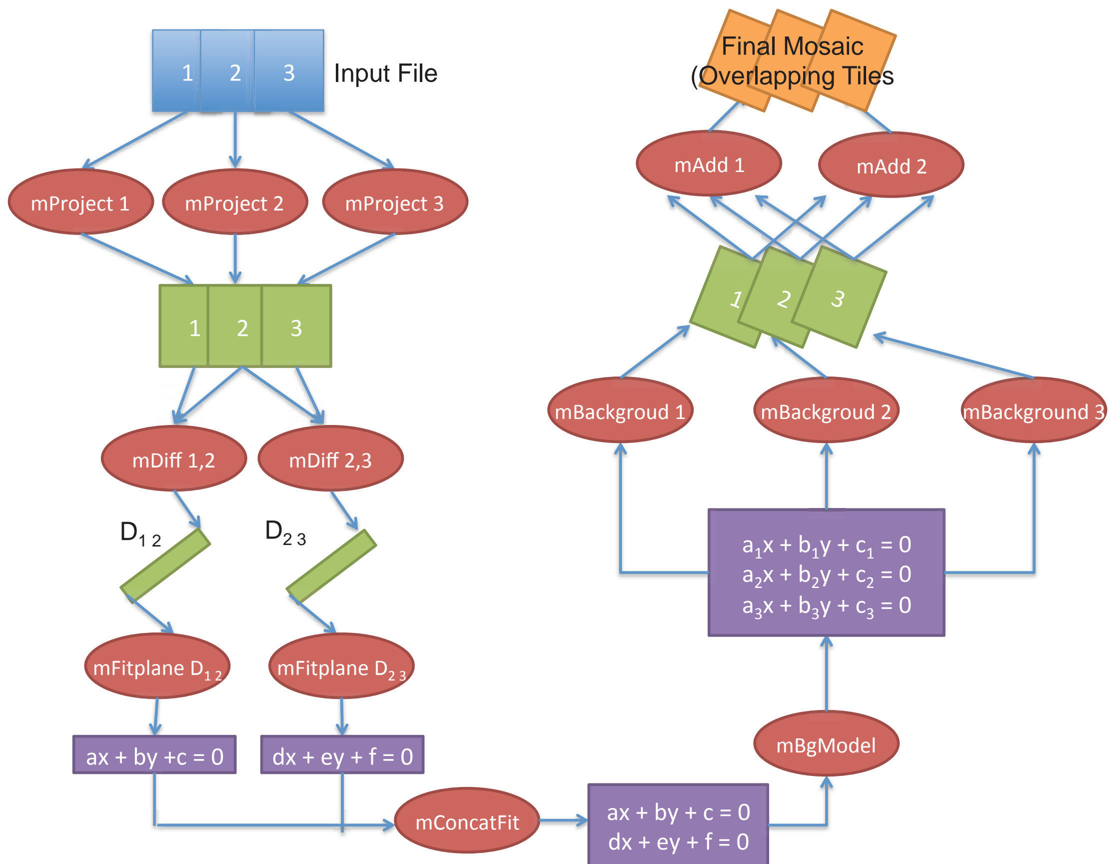


Figure 2.1: The Process of Montage [14]

To investigate the temporal I/O locality of data intensive workloads, we developed MUSE [75] to trace all I/O operations. We used MUSE to determine the I/O locality of Montage with three different datasets. The results of these experiments are shown in Table. 2.1. The buffered I/O ratio, R_{buf}, means the ratio of read and write operations. The ratio can be formulated as:

$$R_{buf} = \frac{r_b + w_b}{r_t + w_t} \tag{2.1}$$

where r_t and w_t denote total read and write sizes, respectively; and r_b and w_b denote size of reading and writing from the buffer, respectively. As shown in the table, we can see that the I/O pattern of Montage shows high buffered I/O ratio, which is over 60% for all the datasets. Because we assume a buffer of adequate size, all written data can be buffered, i.e. $w_b = w_t$.

Large-scale HPC applications also have high temporal I/O locality because these applications usually write checkpoints for fault tolerance. In checkpoint/restart, applications read the *latest* checkpoint when restarting. The I/O pattern increases temporal I/O locality. Thus, improving read and write performance in r_b, w_b can accelerate these data intensive workloads.

2.2 Overview of Cloud Computing

Although traditional HPC centers can provide good environments for data intensive applications, there are several limitations:

- First, only authorized users can have access to the HPC centers.
- Second, the resources in HPC centers are often short for demands, users need to wait in job queue for long time to get their applications executed.

The increasing requires for larger data processing causes the HPC centers can't handle all the requires for users especially unauthorized users, hence new approaches need to be found.

In order to solve the problems mentioned above, and let more users can execute applications at large scale, we consider migrating these applictions to cloud environment. Cloud computing is gathering more and more interests recently. Clouds provide virtually unlimited resources and adopts pay-as-you-go pricing policy makes them greatly attractive for data intensive applications. The key concept of cloud computing is sharing and converging infrastructure over the

Internet. Users allocate resources before they start to use it and deallocate after the job finished, and pay what they used. The same resource can be used by different users at different time point, it increases the utility of underlying infrastructure. All the resources in cloud can be accessed via Internet, users can control and use these resources any time any where. The key characteristics of cloud can be summarized as follow:

- **On-demand**
  Users are able to allocate only what they need, and pay for only what they consumed.

- **Network Access**
  All the resources can be accessed via Internet, it means users can use cloud service any time any where.

- **Resource Pooling**
  The cloud resources are pooled and shared by multiple clients. The resources are assigned and reassigned to clients according to their demands. Clients have no knowledge about the location of resources they are using due to the high abstraction of cloud resources.

- **Rapid Elasticity**
  The resources in cloud can be allocated rapidly and elastically. To customers, the resources are appeared to be virtually unlimited and can be allocated at any time. Some cloud even offers auto scale out, the resources can be increased automatically when the computation peak comes.

- **Measured Service**
  Due to the high level abstraction of resources in cloud, both cloud provides and clients can manage, control, the resource usage, and clients are charged according to their usage.

Cloud computing can be categorized into following three models (Figure 2.2):

- **Infrastructure as a service (IaaS)**
  IaaS model provides the most basic service in three models, vendors provides only the physical or virtual machine and other resources. Users need to install OS and configure the environments by themselves before executing their applications. Example: Amazon Web Service (AWS) [1].

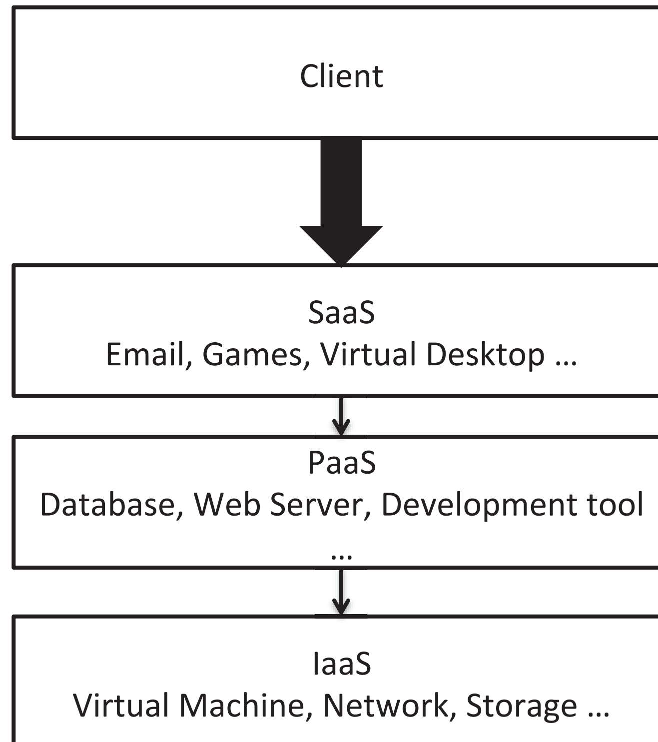


Figure 2.2: Three Models in Cloud Computing

- **Platform as a service (PaaS)**
  In PaaS model, users receive software platform with OS and several other environments. Users can quickly execute their applications on these platform without considering the details of underlying machines. Example: Google App Engine.

- **Software as a service (SaaS)**
  SaaS provides is the highest layer in three models, SaaS provides existing applications directly, providers manage the infrastructure and platform. Users only need to care about how to use these applications. Example: Dropbox.

Currently, clouds are widely used in various fields, more and more companies and organizations join cloud computing. They provide their own cloud service to the public, this is called public cloud; On the other hands, companies set up cloud environments only for internal usage for security concerns or performance concerns, such cloud is known as private cloud; Hybrid cloud is the combination of public and private cloud. Figure 2.3 shows the relationship between these three kinds of clouds.

- **Public Cloud**:
  Public Cloud is provided by public cloud providers via Internet. Users all over

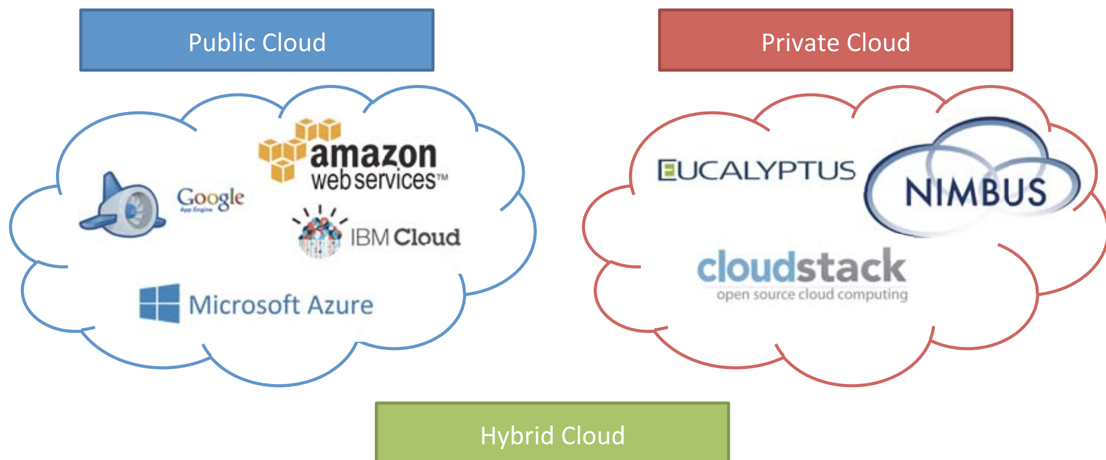


Figure 2.3: Public/Private/Hybrid Cloud

the world share the resources together. Public cloud can be IaaS, PaaS and SaaS, cloud providers take care of the underlying Infrastructure, both hardware and software, and users don't need to care about these issues. Public cloud providers usually adopt pay-as-you-go pricing policy, users pay for what they used. In following situations people would like to use public cloud, When:

- People want to provide services or applications for clients to access from all over the world.
- People need to save their money or efforts on infrastructure.
- Services or applications need incremental capacity for temporary period (increase machine for peak times).

- **Private Cloud**:
  Private cloud, on contrary, can only be accessed from certain network, LAN or via VPN. Private cloud provides higher protection on data and applications since these data can only be accessed internal, and private cloud can easily achieve high performance since local network performs well compared to the Internet as well as less people share the resources compared to public cloud. However companies or organizations need to set up and maintain both hardware and software of private cloud, and pay for the initial costs as well as maintenance costs. The scalability of private clouds is usual limited by underlying hardware and software and cannot scale on-demand. Private cloud is better, when:

  - People need to confirm to strict security and data privacy issues.
  - People need extremely high performance for their applications.

- **Hybrid Cloud**:
  Hybrid cloud is the combination of public cloud and private cloud, provides a solution between them. With hybrid cloud, both the benefits in public and private cloud can be achieved. However the disadvantage is that data and job movement between two environments are required.

2.3 Burst Buffer

Executing data intensive HPC workloads in clouds may result in unacceptable performance degradation due to low I/O performance as well as inefficient file metadata operations in cloud storage (Section 2.4.3). Our analysis of data intensive workloads on HPC systems also found that a number of applications have temporal I/O locality (Section 2.1.2). These I/O workloads can be accelerated even on clouds even if performance of shared storage is low (Section 2.4.3) by using our cloud-based burst buffer system as an *on-demand remote cache space*.

2.3.1 Overview of Burst Buffer

The storage system in high performance computing system was designed to meet the high bandwidth, low latency, high scalability requirements. Currently, high performance applications run on large scale computer system can achieve up to one PB/s for peak, and sustained bandwidth from 40 GiB/s to 80 GiB/s. However recent researches show that applications may perform bursty I/O patterns because they alternate between computationally dominant and I/O-dominant. Furthermore, the requirements for high performance storage systems keep growing. In near future, top-class systems are expected to have 100 times to 1000 times more compute nodes than today's systems. More compute nodes means more parallel jobs and more I/O bandwidth requires.

However, increasing the performance of parallel file system for 100 times creates a great challenge for storage system designer. A traditional approach is to simply increase the number of storage nodes and the cables and other network infrastructures. However, researches warn that such approach will result in significant underutilization of the storage system, since applications only require for the peak I/O performance for a short time. Such underutilization has already been observed in current HPC systems. For example, The I/O performance statistical data collected from Argonne

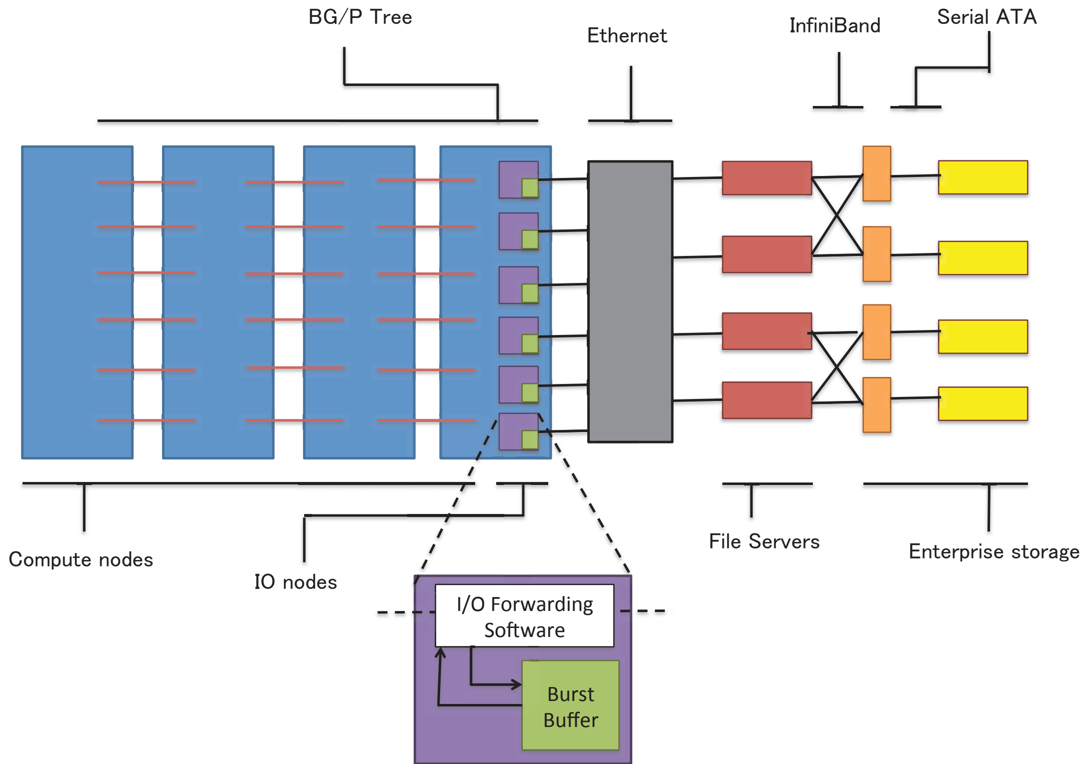


Figure 2.4: Burst Buffer Tier in HPC System [54]

Leadership Computing Facility shows that the typical bandwidth of IBM Blue Gene/P storage system is less than one-third of its capacity 99% of the time [54].

In order to absorb such bursty I/O data from applications in current storage system, burst buffer has been proposed as a new tier of HPC system [54]. Figure 2.4 shows the architecture of HPC system with burst buffer tier. It has been first added into Argonne IBM Blue Gene/P(Intrepid) computing environment. Burst buffers are high throughput low capacity storage devices that act as an intermediate storage layer in HPC storage system. Burst buffers located in a set of I/O nodes, which have a tier of solid-state disk(SSD) and locates around the compute nodes. The burst buffers and compute nodes are connected via high performance network. With such burst buffer tier, bursty I/O data from applications is forwarded to burst buffer, and buffered in SSD memory temporarily with high bandwidth, applications can go back to computation without waiting the data to be finally written back to storage. Through simulations on IBM Blue Gene/P with CODES simulator and I/O patterns of synthetic benchmarks and scientific applications, burst buffers can significantly improve the perceived I/O performance from client side by overlapping computation and writing back from burst buffers to external storage.

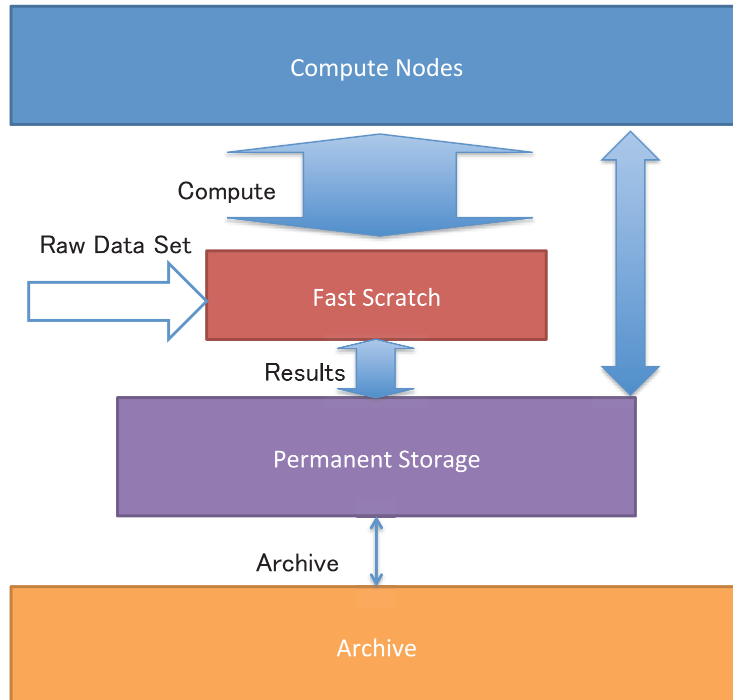


Figure 2.5: Modular HPC Architecture

2.4 Storage System in Cloud and HPC

Although clouds exhibit several characteristics in high performance and scalability, which make clouds suitable for HPC applications, if we compare the storage system, we still find a huge gap in performance between cloud storage and HPC storage. Such performance gap in storage systems has huge impacts to data intensive applications since these applications have heavy I/O burdens. Migrating data intensive applications to cloud with low performance storage systems causes the decrement in performance and the prolonged execution time causes more cost to users due to pay-as-you-go cloud pricing model. In this section, we first introduce the overview of HPC and cloud storage. Then we discuss about the challenges we encounters when migrating data intensive applications to cloud.

2.4.1 HPC Storage System

The requests to process larger data size quickly motivate the evolution of HPC storage system. Unlike traditional file systems, HPC storage systems consist of a number of storage nodes, each storage node has its own local storage and network connection. These storage nodes are clustered to provide high aggregated bandwidth and capacity

to fulfill the requests. HPC storage systems are designed for high performance, high scalability, high capacity and low latency.

- **Scaleable, High IOPS**
  The first request is high IOPS to handle massive I/O requests from nodes in HPC systems. Flash storages are widely used in mainstream data center to increase the IOPS. However, this may not be an option for HPC system since the capacity demands are also extremely high. Currently, the capacities of flash storages are fairly small compared to tradition hard disks and tapes. Flash storage vendors provides techniques like compression and deduplication to increase the space efficiency, but such techniques can not be effective in HPC system since HPC data tends to be highly unique. Flash storage does have been used in HPC system. Burst buffer is a example to use flash storage to increase IOPS in real HPC system. Besides high IOPS, the scalability to expand the IOPS is also critical. Given the number of compute nodes increases in the HPC systems, such scalability to increase IOPS by increasing number of storage nodes is the important to keep pace with the request increment. The key point in IOPS scalability is the distribution of workload. Many efforts have been made to achieve balanced workload in various scenario.

- **Scalable High Bandwidth**
  Another request for HPC storage is to provide enough bandwidth for compute nodes in HPC system. High bandwidth can be achieved from aggregation across multiple storage nodes. Especially for data-intensive applications, which spend major time in I/O, the I/O bandwidth is critical, they have massive data to be read/written within a short time. Like the IOPS, scalable and load balance is key point to keep pace with the increment of HPC system.

Besides of increasing the performance of each storage nodes and increasing the number of storage nodes, current HPC storage system tends to use multiple tiers in storage system. Like cache system in computer architecture, each tier has different capacities and performances. Figure 2.5 shows a typical HPC storage system with multilevel storage hierarchy. As shown in the figure, there are three levels in the system:

- **Fast Scratch**
  This layer provides the maximized throughput and IOPS for applications. Temporary results and recent accessed data are stored in this tier. Although it

has the smallest capacity in the three levels, data isn't intended to be stored in this tier for long term. Frequent data transfer are performed between this tier and permanent storage.

- **Permanent Storage**
  Permanent storage store data for applications' processed data and processing data, and usual serve as /home directories for users. This tier provides a balance between performance, capacity and reliability. This tier is reliable and easy to manage. Data swap out/in between fast scratch and this tier.

- **Archive**
  Archive tier has the maximum capacity, but the lowest performance in all three levels. It is designed as long-term storage for data. Some archive storages require policy engine to automatic move data in and out.

There are commonly three different approaches in HPC storage system design:

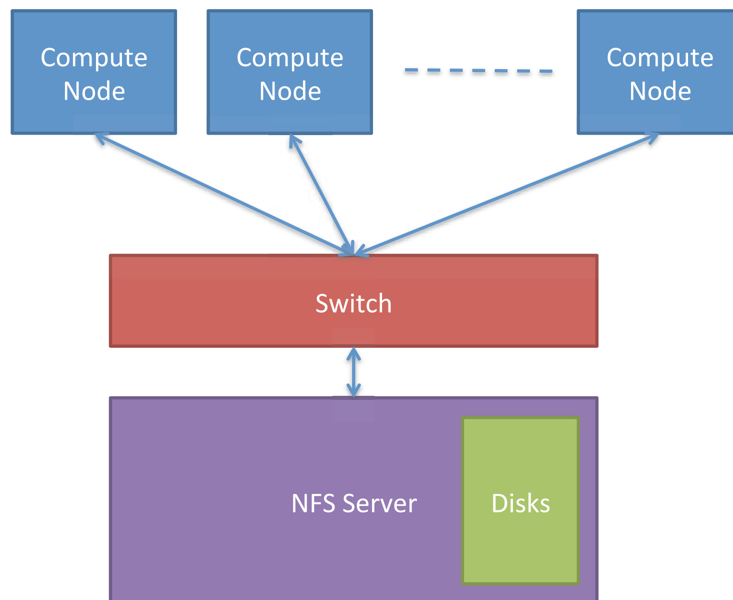


Figure 2.6: Direct Attach SAS Storage

- Direct attach SAS storage
  The first approach (Figure 2.6) is direct attach SAS storage to compute nodes as home directory or application directories. Network filesystem like NFS/CIFS are usually used as shared storage systems. Such system is easy to deploy and manage, also cost efficient.

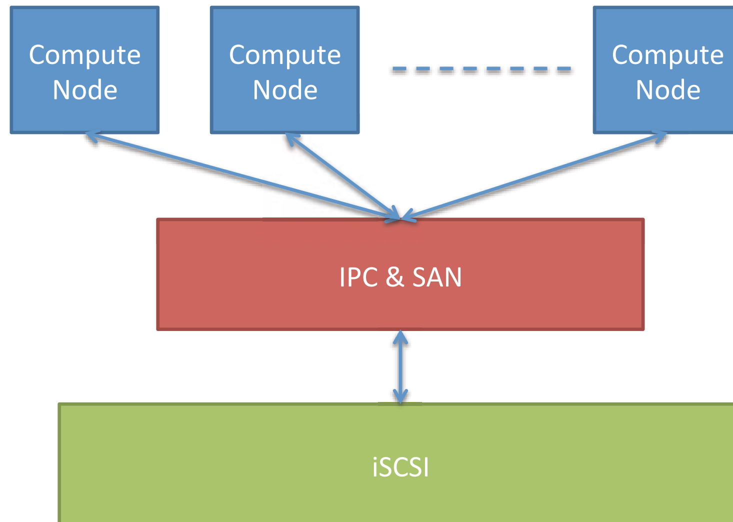


Figure 2.7: SAN Storage

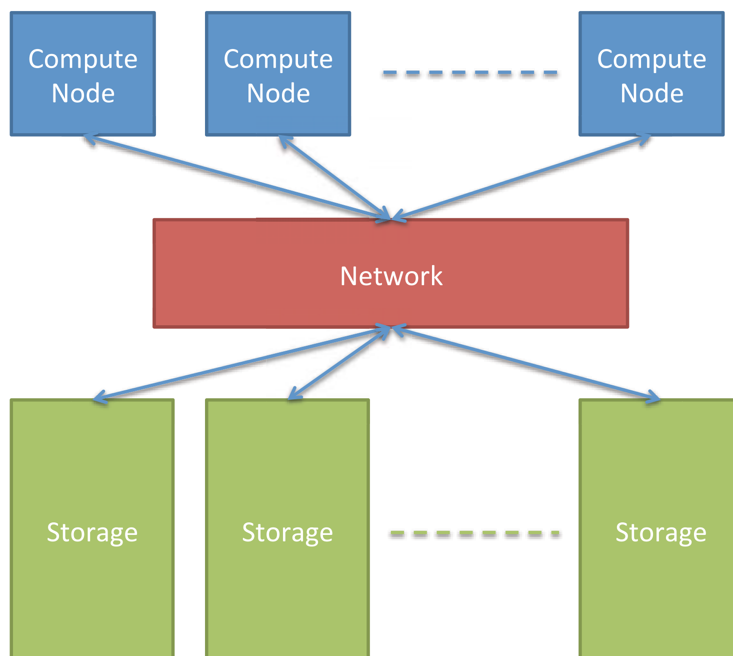


Figure 2.8: Parallel I/O Subsystem

- SAN
  The architecture of second approach (Figure 2.7) is similar to the first approach. The difference is that in first approach, filesystems like NFS hide the details of underlying physical storage, and provide high abstracted data accesss, on the other hand, SAN allow direct block storage access. SAN provides higher performance than SAS, hence SAN can be used in

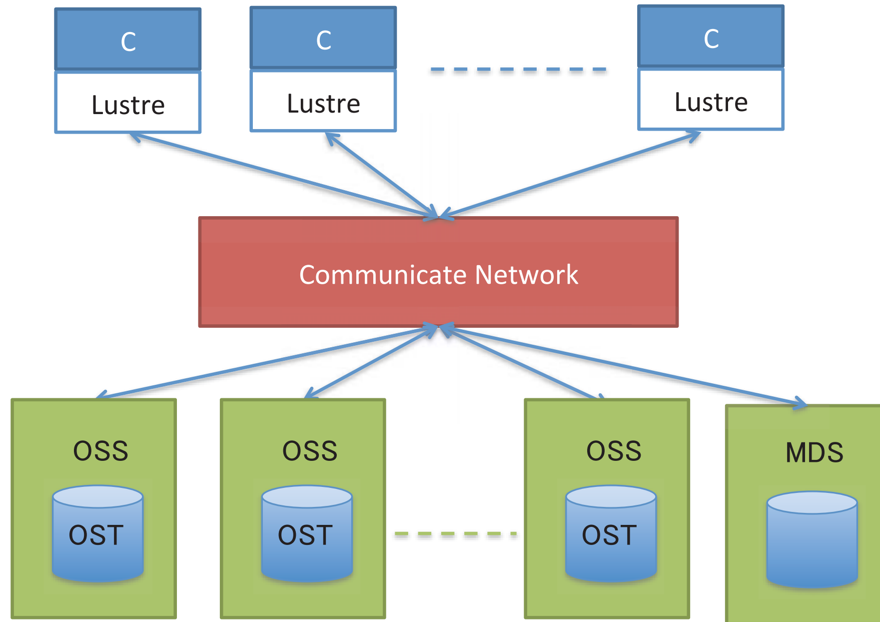


Figure 2.9: Lustre Architecture

small/medium cluster. However the scalability of SAN is still limited and the cost for SAN is higher than simple SAS.

- Parallel I/O Subsystem
  In the third approach (Figure 2.8), multiple storage nodes are used to increase the aggregated throughput and capacity. Multiple compute nodes access to the storage system simultaneously, hence the system exhibits high performance scalability. Such systems are appropriate for large cluster requires for high performance parallel I/O and collective I/O operations. However the increased complexity increases the deployment and management cost.

The third approach is widely used in large HPC systems for its high performance, capacity and scalability. In this paper we focus on this approach. There are many sophicated and famous HPC storage systems. Here we introduce some of them:

- Lustre

  Lustre is one of the most famous parallel distributed file systems under Linux. It has been introduced since Dec, 2003. The name is derived from Linux and cluster. Figure 2.9 shows the architecture of Lustre. The whole system consists of clients, one or more meta data servers (MDS), object storage servers(OSS), each OSS manages one or more local object storage targets

Feature	Current Practical Range
Client Scalability	100-100000
Client Performance	Aggregate: 2.5 TB/sec
OSS Performance	Aggregate: 2.5 TB/sec
MDS Performance	35000/s create operations, 100000/s metadata stat operations

Table 2.2: Lustre Performance [13]

(OST). MDS maintain the namespace and file meta data, the actual data is stored in OSTs . Clients run users' applications and access the data from OST using a POSIX standard UNIX file system interface. Unlike other systems, the meta data servers in Lustre only handle the file meta data, they are not involved in any I/O operations to avoid becoming bottleneck in I/O operations. When client access to a file, it first look up filename on MDS, and then connect to the corresponding OST for data transfer. Clients don't directly modify the data in OST, they issue requests to OST. Such approach enables high scalability and increases the security and reliability. Table 2.2 shows the practical range performance of Lustre.

- PVFS

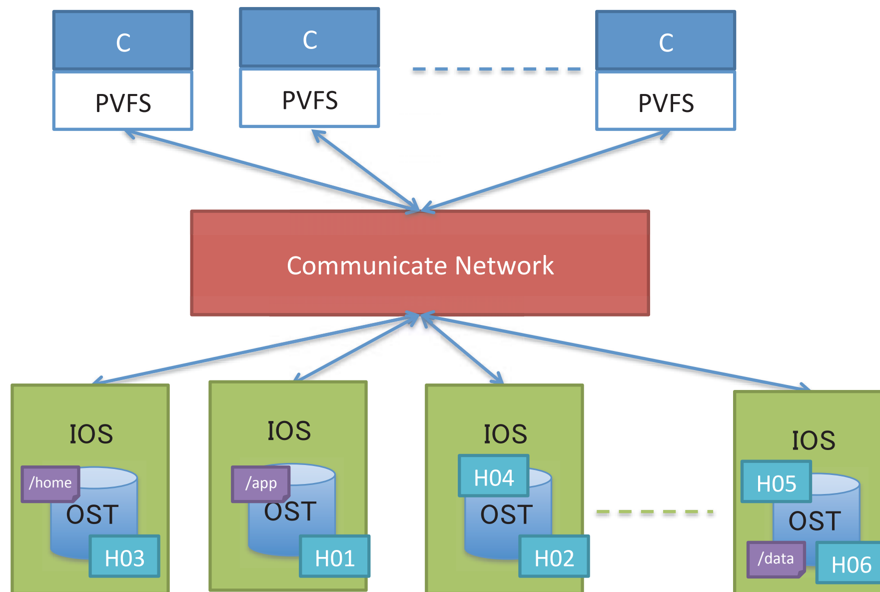


Figure 2.10: PVFS Architecture

inode	134135
⋮	⋮
base	1
pcount	3
size	65536

Figure 2.11: An example of File Block List in PVFS

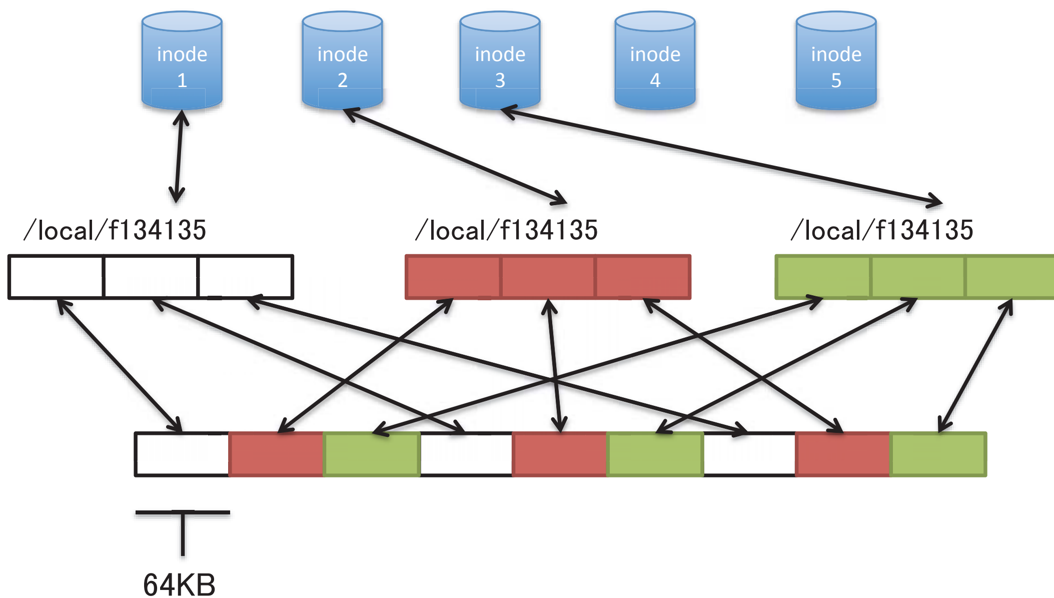


Figure 2.12: PVFS File Striping Example

PVFS is an open-source parallel filesystem. Like Lustre, PVFS was also designed for large clusters running Linux. PVFS consists of a server process and a client library. Clients can mount PVFS via a Linux kernel module and pvfs-client process. Figure 2.10 shows the architecture of PVFS, the whole system consists of PVFS clients, I/O manager and I/O nodes. I/O managers maintain the file meta data and I/O nodes store the actual data.

PVFS split files into several blocks and distribute these blocks among I/O nodes. I/O managers maintain the list of file blocks as Figure 2.11 shows. Field *base* denotes the id of I/O nodes which stores the first block, here I/O node 1 store

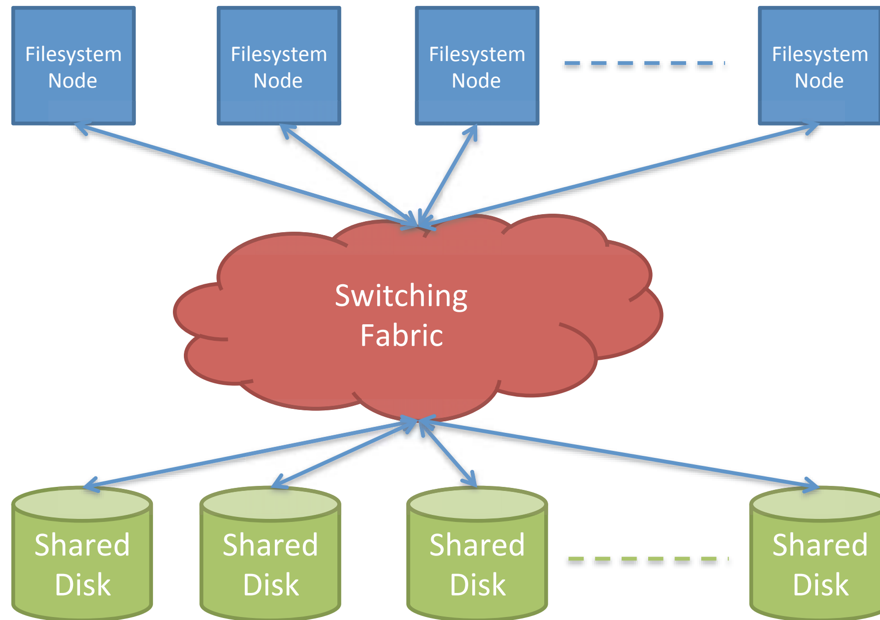


Figure 2.13: GPFS Architecture

the first block. Field *pcount* means the number of I/O nodes on which blocks distributed, here blocks distributed across 5 I/O nodes. Field *size* shows the size of each block, which in this case is 64KB. Hence the file is split into 5 64KB blocks and stored on 5 different I/O nodes from I/O node 1 to I/O node 6. Figure 2.12 shows the file striping details.

In order to accelerate the data transfer, clients communicate with I/O manager for the file meta data information only before I/O operations, after getting the file meta data and file block list, clients connect to corresponding I/O nodes directly for data transfer. PVFS supports three kinds of interface: POSIX; Unix shell interface like ls, cp; MPI-IO.

- GPFS
  GPFS (General Parallel File System) is a high-performance parallel filesystem developed by IBM. GPFS achieve high performance by allowing multiple concurrent access. Figure 2.13 shows the architecture of GPFS, the whole system consists of filesystem nodes and shared storages, network like SAN are used to connect them. Like Lustre and PVFS, GPFS also strips blocks of data from a file, and distributes blocks among storage nodes to avoid concentrated access on single node. GPFS split files into fixed size blocks, and assign shared storage to store blocks according to round-robin algorithm. The fixed block size can set to 256KB to improve the seek time, or set to 1MB 16MB to

improve the I/O throughput. In order to fully utilize the network, GPFS implemented I/O buffers for pre-fetch, and these buffers can be filled in parallel.

2.4.2 Cloud Storage System

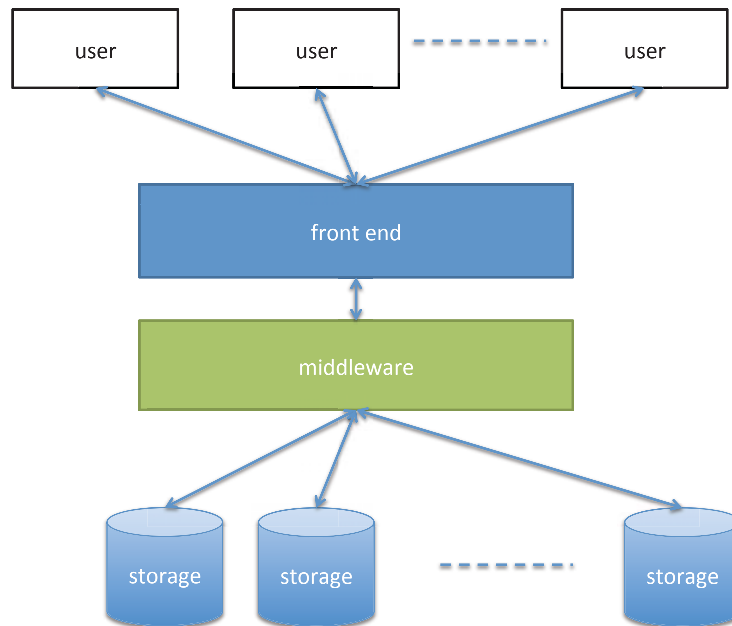


Figure 2.14: Cloud Storage Architecture

First we take a look at cloud storage system. As a part of cloud computing, cloud storage is also highly virtualized, and can be accessed through Internet. Users share the resources in the resource pool and pay for what they consumed. Cloud storage offers users ability to meet demands of increasing data volumes, reducing cost for hardware and security issues. There are hundreds of cloud storage system, some are for special proposes and others are for general propose. Sophisticated cloud storages consist of thousand and hundreds of machines and disks, it is because they are shared by a large amount of users simultaneously, and they need to provides high performance as well as high storage capacity for each user. Furthermore, a system with large number of machines and disks need maintenance or repair frequently, and machine or disk failures also occurs frequently. It is important to store the same information on different storage nodes in order to ensure data access when some machines and disks are under maintenance or repair and prevent data loss when machine or disk failure occurs. This is called **redundancy** or **replication**. Without

such mechanisms, cloud storage can not ensure data access at any time point, and get rid of data loss due to machine failure. Such huge systems take large space to hold them, the house used to hold large cloud storage system is called **data center**. Users send data to cloud storage server via Internet, and server record these data. When users want to retrieve these data, they access to cloud storage server again, and receive data via Internet.

Figure 2.14 shows common architecture of cloud storage system. There are three lays:

- **Front End**
  Front end consists of several servers, hide the details of underlying infrastructure, and provide interfaces to users.

  Users need to interact with these front end servers to get access to the actual data. There are many kinds of front ends: web service front ends, file-based fronts end or more traditional front ends like Internet SCSI.

- **Middleware**
  Between front end and physical storage, there is middleware control the logic management of data. Middleware controls algorithm like duplication, data reduction and data placement.

- **Physical Storage**
  Physical storage store the actual data. There are many different architectures for the physical storage system.

There are many companies and organizations provide their cloud storage service over the Internet. Here are some famous cloud storages:

- **Google doc**
  Google doc let users to upload and edit documents, spreadsheets and presentations to their server. Users can access Google doc by their web browser or using a Google application.

- **Gmail**
  Users can access to their mailbox from their computers or mobile devices via Internet. There are many other e-mail providers like yahoo and hotmail.

- **Youtube**
  Video sites like Youtube let users to enjoy videos and audios any time and any

where. Youtube stores millions of user-uploaded videos. There are also many alternatives like youku and niconico video.

- **Facebook**
  Social network sites like facebook let users to post their status, messages and pictures. These messages can be shared among their friends. Here are some other social network sites like twitter.

Cloud storage systems have many different implementation to fit for different requirements. Followings are common types of cloud storage:

- **Object Storage System**
  Object storage system is much simpler than traditional file systems, and hence is more scalable. Instead of managing files in directory hierarchy, object storage stores file into special containers, and assigns with a unique id, files can be retrieved through these ids. Object storage system also stores the metadata into the same container, so that the overhead of metadata access can be distributed among the storage nodes, and the whole system can scale out by adding storage node. Reliability can be achieved by add replicas of files. However creating more replicas becomes heavy overhead to the system, so the number of replica is usually configured by the providers to achieve the best balance between reliability and performance.

  The drawback of object storage system is also because such replication, object storage system only guarantee eventual consistency. Users may read stale data before all the replicas are updated, even after several successful reads, users must tolerate such inconsistent read. Such characteristic makes object storage not suitable for frequent data update.

- **Block Storage System**
  Most local file systems are block storage. Unlike object storage system, block storage systems divide files into evenly size *data blocks*, and read/write for the whole block at a time. Each block has it address, and stores only actual data, users can retrieve the data according to the address. Special blocks(known as *inodes*) contain meta data of files such as file name, file size, and maintain list of *data blocks*. Block storage system can not allocate size less than block size, it cause the space inefficiency(known as *internal fragmentation*) when the file size is not integer multiples of block size. The last blocks in file block list are partially empty in such cases. There is a trade-off in block size decision. Since

block storage read/write a whole block at a time, larger block size leads to more sequential I/O and hence better performance. However due to *internal fragmentation*, larger block size also causes more space waste. The common block size in local file systems is 4KB since the overhead for seeking block is fairly small in local file system, hence small block size is applied to save space. On the other hand, the block sizes in remote block file system are usually from several to several hundreds MB to improve the performance.

- **Relational Database Storage Systems**
  Cloud database is a kind of database run on cloud. There are two kinds of cloud databases: databases created and managed by users themselves, and databases maintains by cloud providers. Cloud database reduce the burden of configuration, scaling, access control, performance tuning, backup from users. Users can focus more efforts on their projects, and the cost to users is much lower because they pay for a shared service instead of creating their own one.

- **Distributed File Storage Systems**
  Distributed file storage systems allows multiple users to access the same files. Files are usually split into several data chunks, and stores in several storage nodes. The clients don't have access to the underlying block system directly, but they interact over a certain network protocol, data is sent/received on network. The distributed system provides high availability and high scalability.

The biggest concerns when using cloud storage is **security** and **reliability**. Users, especially for companies, data security is the main concern, they must ensure that there is no unauthorized data access. In order to secure data, following technologies have been introduced into common cloud storage system.

- **Encryption**
  Encryption means data was encrypted with special keys by a complicated encryption algorithm like AES or RSA, before uploaded to cloud storage. When users retrieve their data back, they need to decrypt data with corresponding key. Encryption can protect data even there is any unauthorized download.

- **Authorization**
  Authorization requires a user list for authorized clients, and only these users have access to data. Many companies have multiple-level authorization. Each

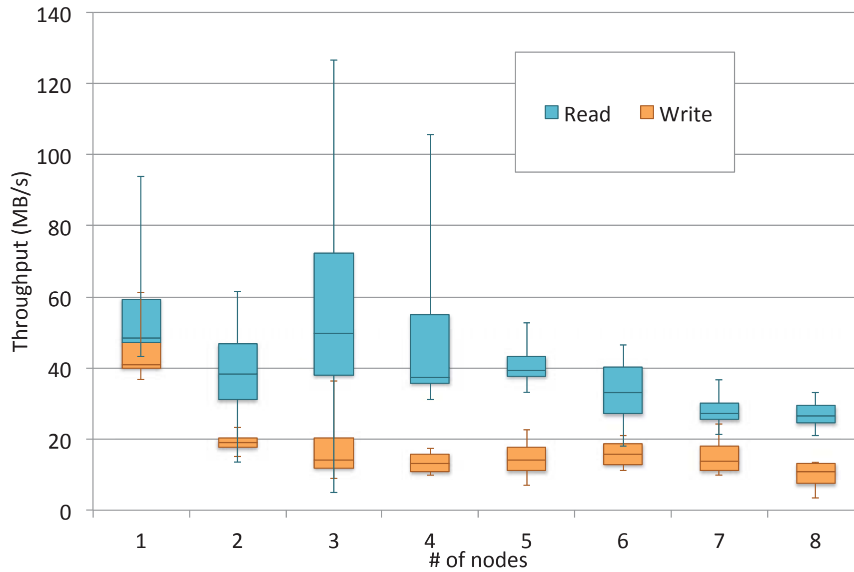


Figure 2.15: Read/write I/O throughput with a single shared file on Amazon s3.

level represent different permissions for data access. For example, Low level users have limited access to data, on contrary, high level users have full access.

2.4.3 Problems in Cloud Storage Systems

There are two major challenges in data-intensive applications migration from HPC centers to clouds. The first one is performance issue, as we discussed in previous sections, the storage systems in HPC centers were designed for high scalable throughput and low latency. However the storage systems in cloud were designed for high availability and high capacity. Hence there is a huge performance gap between storage systems in HPC centers and clouds. Moreover, cloud resources are sharing by multiple users, which further exacerbates the storage performance in cloud. Besides the performance issue, consistency policies adopted by cloud providers also have large impacts on applications migrations. Current cloud providers adopt a relaxed consistency policy called *eventual consistency* in favour of improved performance and latency. Such relaxed consistency causes inconsistent read when applications share data over multiple nodes, and consequently leads job failure. In the following of this section we show these two issues in cloud storage systems in detail, and then we introduce our solutions to these issues.

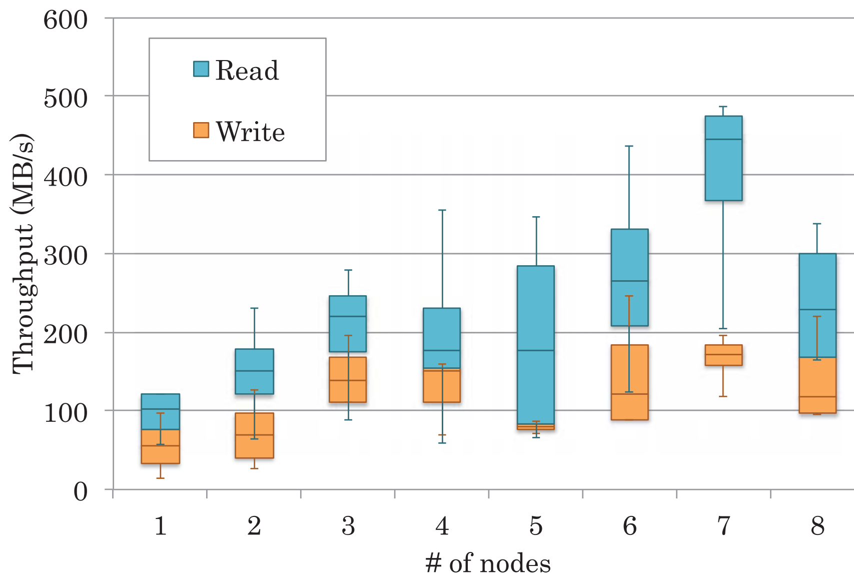


Figure 2.16: Read/write I/O throughput with each nodes accessing its own file on Amazon s3.

IO Performance in Cloud Storage Systems

Figure 2.15 shows I/O throughput with N number of nodes concurrently reading or writing to a single file on the Amazon S3 cloud. In the best case, i.e., reading using 3 nodes, the maximum I/O throughput is only 150 MB/s. In a different experiment with N compute nodes accessing to its own individual file, we see improvements in I/O throughput and scalability as shown in Figure 2.16. However, the improvements are still limited compared to state-of-the art parallel file systems, especially where write performance does not scale with the number of nodes. From the figures, we also see that the I/O performance is fairly unstable. Because typical data intensive applications consist of processes with mutual dependencies, prolonged I/O operation due to this instability can propagate, degrading the overall performance [14]. Also, not only I/O throughput but also metadata operations, such as file creation or get file status etc., can become bottlenecks of I/O performance.

Eventual Consistency Policy in Cloud Storage Systems

Modern clouds provide NoSQL storage substrates such as Amazon SimpleDB [3]; such storage, and the underlying storage in their implementation as with Amazon S3, typically sacrifice consistency in favor of improved latency for performance, as well as availability, using relaxed consistency protocols such as *eventual consistency* [26].

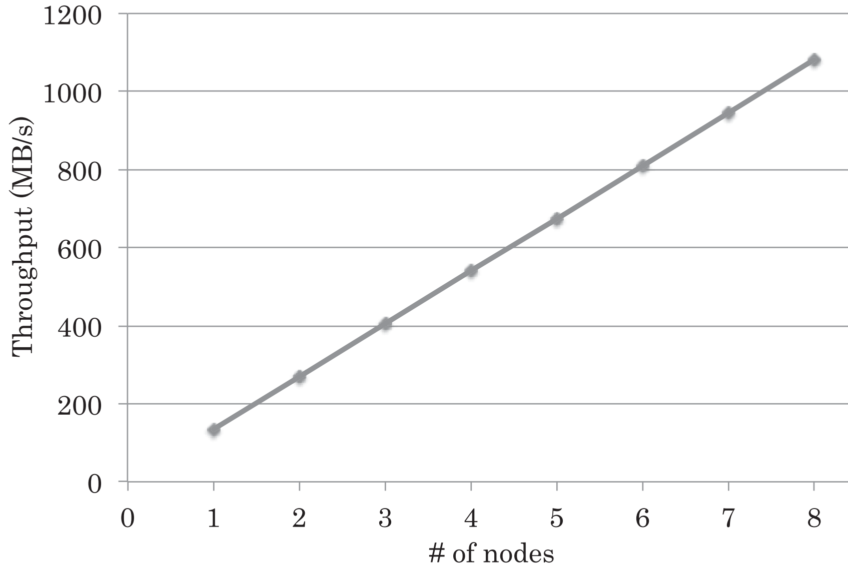


Figure 2.17: Point-to-point communication throughput in Amazon EC2

Although they are fine for standard cloud workloads, for workflows of data intensive HPC workloads, where the result from the preceding nodes in the workflow is passed onto the succeeding nodes under the assumption that there is a high-performance and consistent filesystem such as Lustre, execution in a cloud environment could result in an error as stale data may be read [39]. This problem has been reported in various studies [30, 37, 40, 67, 88], and so far the solutions have been to employ stricter consistency models which sacrifice performance. As such, we need new methodologies for achieving high performance in data intensive HPC workflows in clouds, and our cloud burst buffer is designed exactly to cope with the problem.

Motivation to Burst Buffer in Clouds

As mentioned in Section 2.1.2, data intensive workloads with high temporal I/O locality stand to benefit from their I/O data being buffered. If we can install *large*, *fast*, and *shared* buffer spaces, we can accelerate these data intensive workloads. Motivated by this fact, we enhance the performance of the storage hierarchy of cloud environments though the use of a *burst buffer* technology. Burst buffer is a new tier in current storage hierarchy for bursty I/O operation in data intensive applications [54] and checkpointing workloads [74]. By incorporating the new tier of storage into clouds, the bursty I/O workloads can be absorbed without a need of higher bandwidth storage.

Figure 2.17 shows point-to-point communication throughput in Amazon EC2. In this evaluation, we measure the network throughput between a pair of $m3.xlarge$ instances using Iperf [12], and increase the number of the pairs. The specification of the instances is in Table 3.3. As shown in the figure, we can see that the network throughput between two nodes is high and proportional to the increasing number of the pairs. We take advantage of the high and scalable network throughput to burst I/O throughput. If we use the combined memory space from a group of instances as burst buffers, we can construct on-demand burst buffers with high throughput and high scalability on the fly.

2.5 FUSE

In order to implement our proposal burst buffer system, we need to forward applications' I/O requests from compute nodes to burst buffer nodes. Furthermore, in order to reduce the modifications in source code when users migrate their applications from general UNIX system to our burst buffer system, we need to support general I/O APIs in our system. For these reasons we chose to implement our system with FUSE framework. In this section, we give a brief introduction about FUSE framework.

FUSE is the abbreviation of Filesystem in Userspace, which is a framework to implement users' own fully functional filesystem. Figure 2.18 shows the Architecture of FUSE framework. As we can see from the figure, FUSE framework redirect API calls from glibc to user-defined handler functions, and return the result from these handler to applications. With FUSE framework, users' filesystem can act as normal file system. FUSE has following features:

- Simple library API
  FUSE provides several hooks (Table 2.3) and filesystem developers only need to implement some of these hooks to create their own filesystem.

- Simple installation
  There is no need to patch or recompile the kernel when using FUSE.

- Efficiency
  Userspace - kernel interface is very efficient.

- No-root
  Usable by non privileged users.

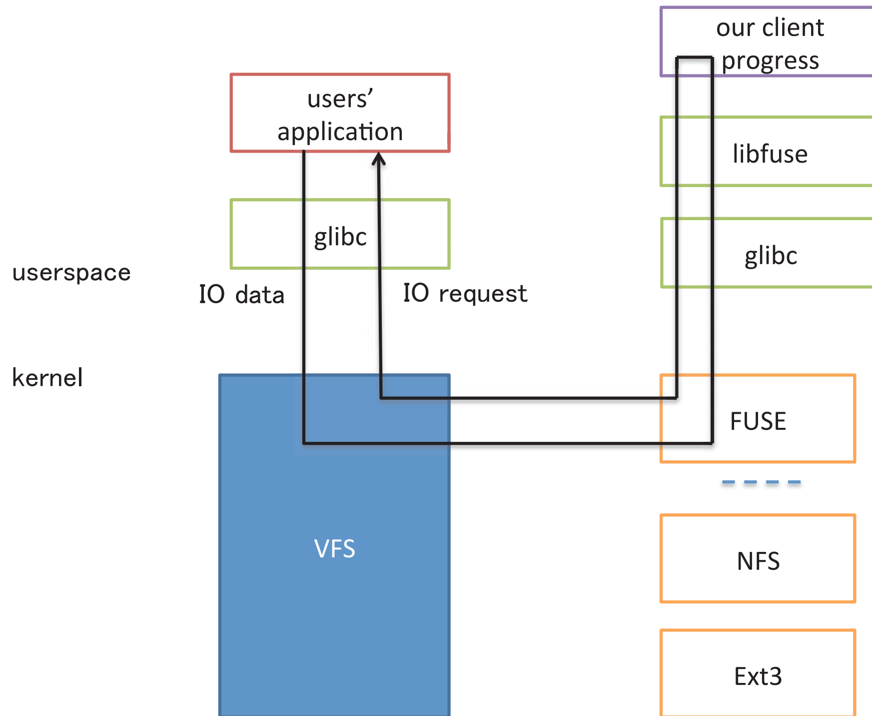


Figure 2.18: FUSE Architecture

- Safe
  Secure implementation.
- Stability
  Has proven very stable over time.

continued from previous page	
getattr	Get file attribute, similar to stat()
readlink	Return the target of a symbolic link, similar to readlink()
mknode	Create all non-directory non-symbolic link nodes, if filesystem defined create function, that that will be called instead
mkdir	Create a directory, similar to mkdir()
unlink	Remove a file, similar to unlink()
rmdir	Remove a directory, similar to rmdir()
symlink	Create a symbolic link, similar to symlink()
rename	Rename a file, similar to rename()
continued on next page	

continued from previous page

link	Create a hard link to file, similar to link()
chmod	Change the permission of a file, similar to chmod()
chown	Change the owner of a file, similar to chown()
truncate	Change the size of a file, similar to truncate()
open	Open a file, similar to open()
read	Read data from an open file, file must be opened by O_RDONLY or O_RDWR, similar to read()
write	Write data to an open file, file must be opened by O_WRONLY or O_RDWR, similar to write()
statfs	Get file system statistics information
flush	Flush buffered data associated to a file, similar to flush()
release	Close an open file, similar to close()
fsync	Synchronize file data, similar to fsync()
setxattr	Set extended attributes to a file, similar to setxattr()
getxattr	Get extended attributes associated to a file, similar to getxattr()
listxattr	List the extended attributes associated to a file, similar to listxattr()
removexattr	Remove extended attributes from a file, similar to removexattr()
opendir	Open a directory, similar to opendir()
releasedir	Leave a directory, similar to closedir()
fsyncdir	Synchronize directory data
init	Initialize filesystem
destory	Clean up filesystem, Called on exit
access	Check file permission, similar to access()
create	Create a new file, similar to create()
ftruncate	Change size of an open file, similar to ftruncate()
fgetattr	Get attributes from an open file
lock	Perform POSIX lock operation on a file, similar to flock()
utimens	Change the access and modification time of a file, similar to utimensat()

continued on next page

continued from previous page	
bmaps	Map block index within file to block index within device, only used for block device
ioctl	Manipulates the underlying device parameters of a file, similar to ioctl()
poll	Poll for multiple I/O events, similar to poll()
write_buffer	Write data to buffer associated with a file
read_buffer	Read data from buffer associated with a file
flock	Perform BSD lock operation on a file
fallocate	Allocate space for an open file

Table 2.3: Hooks in FUSE 2.9.4 [36]

FUSE has been used to develop many famous filesystems:

- **sshfs**
  sshfs is a filesystem client based on SSH protocol. Any machine can use SSH can use sshfs to transfer files so it is very easy to set up and can be widely used. Mounting the filesystem on client side is as simple as logging into the server with SSH.

- **HDFS client**
  HDFS, which stands for Hadoop distributed file system, is a distributed and scalable filesystem for the Hadoop framework. HDFS was designed to stored large files across multiple machines and support Map-Reduce operations on Hadoop. The data transfer on HDFS is based on TCP/IP protocol, and clients communicate with each other via remote procedure call(RPC). HDFS can be mounted directly through a client based on FUSE.

- **s3fs**
  s3fs was designed to mount Amazon S3 bucket as local filesytem on any Unix like system through FUSE. Any operations on s3fs can be transparently performed on Amazon S3 bucket. s3fs provides a easy way to read/write Amazon S3 bucket as normal file.

- **gfarm2fs**
  gfarm2fs is a FUSE based client to mount Gfarm filesystem as local filesystem. Gfarm is a open-source network shared filesystem, developed by tsukuba

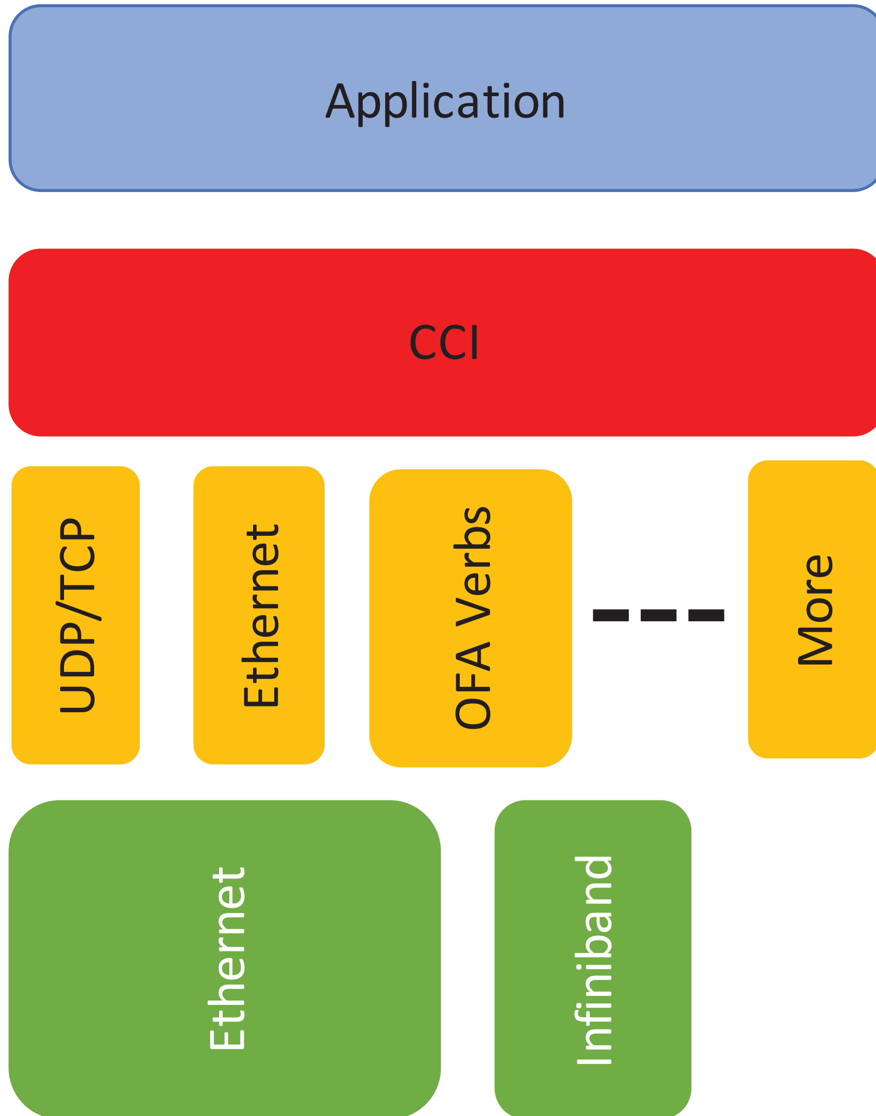


Figure 2.19: Overview of CCI

university and AIST. Tt is generally used in large scale cluster computing. The name, Gfarm, is derived from the Grid Data Farm architecture.

2.6 CCI

CCI [5] (Common communication Interface) is an open-source framework for high performance communication. CCI provides a simple and portable API while ensures high performance, scalability for large deployments and robustness. It is build and maintained by research, industry and academic collaboration.

As a framework targeting on high performance computing center as well as data centers. CCI provides a common network abstraction layers (NAL) to generalize the communication APIs for different kinds of networks. By using NAL in CCI, the same code base can be executed on different network devices without modifications while maintaining high performance.

CCI supports a wide range of networks devices: Ethernet with (UDP and TCP), Verbs with RDMA, Inifiniband, and Cray Gemini.

2.7 Key Configurations in RSBB

When it comes to efficient use of RSBB systems, the most important aspect is how much data applications can directly write to and read from the fast RSBB instead of a slow PFS. Therefore, efficient file stage-in/out between a RSBB and a PFS is critical. However, the traditional RSBB systems only provide file-level stage-in/out functionalities [32]. In this case, whenever an application stage-in/out, files are moved between a RSBB and a PFS in their entirety, even if the portions of the files being accessed are relatively small. This coarse granularity is not efficient, whereas chunk-level swap-in/out is more promising. In the chunk-level approach, files are divided into chunks and accessing a file involves only swapping the requested chunk to the RSBB instead of swapping the entire file. Therefore, the chunk-level swap-in/out is expected to be more efficient than file-level stage-in/out. However, how much this chunk-level approach improves I/O performance is not an obvious question. Furthermore, the chunk-level approach requires additional concerns to be given to the allocation size and the data replacement algorithm.

2.7.1 Allocation Size

The first concern is how the allocated size of the RSBB impacts the application's performance. Under the chunk-level approach, a RSBB works as cache for a PFS and moves chunks between the RSBB and the PFS. If a larger RSBB size is allocated, the I/O performance of the application is expected to be improved since more data can reside in the RSBB and less swapping should be required. However, this can lead to underutilized buffer if excessive amounts of buffers are allocated and remain unused. On the other hand, if the application is allocated an inadequate buffer capacity, frequent swap-in/out can occur. This will result in deterioration of the I/O performance of the application. Moreover, the performance of I/O-intensive

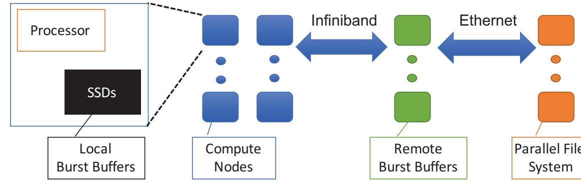


Figure 2.20: An example of burst buffers in an HPC system

applications also depends on the temporaspatial I/O locality of the applications since these affects the I/O access pattern and potential for swapping when the buffer size is insufficient. Given an I/O-intensive application, allocating the appropriate buffer size is not easy.

2.7.2 Data Replacement Algorithm

In CPU cache replacement algorithms, the most common policy is least recently used (LRU). However, in data replacement algorithms for a RSBB, there is no common policy for I/O-intensive applications. Therefore, understanding how different replacement algorithms impact on the applications' performance is important for RSBBs.

2.7.3 Summary

Existing studies have shown that RSBBs can accelerate I/O-intensive applications, which are characteristically slow when only using a PFS [49, 66]. Although different RSBB configurations produce different impacts on the performance of applications under the chunk-level approach (More details in Section 5.4), there are very few studies exploring the various configurations of burst buffers in this manner.

Unfortunately, there are no production RSBB systems that support configuring buffer sizes, data replacement algorithms, and throughputs under chunk-level data management. Therefore, to gain the insight that is required to answer the key questions and support the design of future burst buffer systems, we explore the configuration space through the use of a simulator.

2.8 The Configurations of Burst Buffer

As we mentioned in the Section 1, performance growth of parallel storage systems in HPC centers has stagnated relative to the bursty growth of computational power.

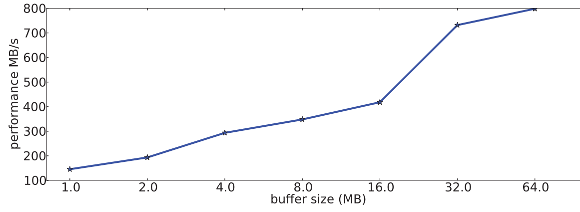


Figure 2.21: The I/O performance under different sizes of burst buffers

Moreover, the compute nodes perform burst I/O from time to time, which causes congestions and requires extremely high bandwidth to handle [54]. However, such burst I/O only lasts for a short time, and designing the PFS to fulfill such requests will cause the PFS to be underutilized most of the time. Burst buffer systems that provide higher throughput but smaller capacities are designed to be used between the compute nodes and the PFS as buffer layers for I/O from the compute node, thereby filling the performance gap [25]. Figure 2.20 shows an example of burst buffers in an HPC system . Several previous studies have shown that by using the burst buffer systems, data-intensive HPC applications which suffer low throughput on the PFS can be greatly accelerated [49, 66, 84].

However, there are very few studies on the configurations of burst buffer systems, such as the buffer size, and the throughput. Like caches in CPUs and other buffers, different configurations of a burst buffer can have great impact on the performance of applications. Hence, configuring the burst buffers according to the system and application requirements are critical. In this paper, we focus on one of the most important aspects in the configurations of burst buffer, the size of burst buffers. With different buffer sizes, performance can vary greatly due to data swapping . Having inadequate burst buffer would seriously degrade the I/O performance, which would cause the I/O becoming a serious bottleneck of the application. In the worst case, applications can perform even worse than having no buffers (Section 5.4). On the other hand, with more buffer, the I/O performance of the applications might be improved, but it can cause underutilization of the burst buffer. For a system with limited amount of a burst buffer, allocating more buffer size means fewer jobs can be run at the same time, and reduction on the total job throughput of the system. Moreover, increasing the buffer size also means increasing of the acquisition cost of the system. Hence the system designers and users are straggling with finding the proper configurations for their burst buffer usage, and our research on I/O analysis and tracing/simulation techniques give key insights to this research area.

Chapter 3

Utilizing Cloud for Large-scale Data-intensive Applications

3.1 CloudBB Architecture and Implementation

Motivated by the problems and the demands for an additional storage tier, we propose CloudBB. CloudBB creates a two-level storage hierarchy by burst buffer techniques, which is a new tier of storage hierarchies for bursty I/O operations in HPC systems. Figure 3.1 shows the overview of CloudBB. Compute nodes (*CNs*) are nodes where applications run. *Burst buffers* are the level-1 storage. The burst buffers are built by another set of nodes in the cloud, and provide remote memory buffers. To achieve scalable I/O operations while maintaining data consistency, CloudBB supports multiple metadata servers in the level-1 storage (Section 3.1.1). The *shared cloud storage*, the existing persistent data store such as Amazon S3, is the level-2 storage By caching popular files on the level-1 storage, CloudBB can significantly accelerate I/O operations to the shared cloud storage (Section 3.2.7).

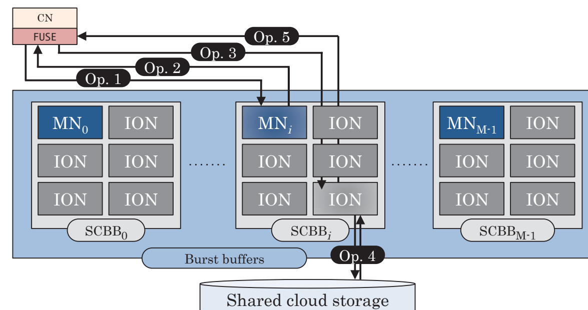


Figure 3.1: Architecture of CloudBB

CloudBB also creates replicas for each file in order to recover files on a failure in level-1 storage (Section 3.1.3). As mentioned, our target is to execute HPC data-intensive applications on clouds and help those who cannot access resources in HPC centers to run applications in a reasonable time. Providing non-standardized I/O interfaces forces users to rewrite applications' code, which is a heavy burden for them, especially for huge applications like data-intensive HPC applications. For this reason, we implement the CloudBB filesystem using `FUSE` [36], existing applications can seamlessly run without modifying applications' code (Section 3.2).

3.1.1 Scalable Multi-Master-Worker Architecture

In HPC systems, PFS needs to provide high consistency because multiple processes create, write, update and read concurrently to shared files. Therefore, PFS has a metadata server and multiple data store nodes, i.e., a Master-Worker model [13]. However, the metadata server can easily become a bottleneck at large-scale. On the other hand, shared cloud storage such as Amazon S3 employs *object store*, i.e., a Key-Value mode. Although object store is scalable, the architecture has data inconsistency problems for data-intensive HPC applications as described in Section 2.

To achieve scalability as well as data consistency, we propose a hybrid architecture of a Master-Worker and a Key-Value model, i.e., Multi-Master-Worker. Figure 3.1 shows the architecture of the hybrid model. The model consists of multiple sub-CloudBBs (*SCBBs*), and a single SCBB consists of a single *Master Node* (*MN*) and multiple *IOnodes* (*IONs*). In our model, multiple MNs manage file metadata in a distributed manner so that a single MN does not become a bottleneck while maintaining consistency within its own SCBB.

An MN is in charge of managing file metadata and controlling all the IONs belonging to the same SCBB. In order to avoid the bottleneck problem, I/O requests from CNs are distributed across different SCBBs by using a hash function. When a CN issues an I/O request, the CN computes a hash value from the file path of the requesting file (`file_path`). For example, given a set of SCBBs (SCBB_0, SCBB_1, ..., SCBB_{M-1}) and `file_path`, SCBB_i is selected by the CN such that $i = \texttt{hash_function}(\texttt{file_path}) \bmod M$. M is the total number of SCBBs. By using this architecture, we are able to hide ION from users, enabling us to dynamically change the number of IONs according to the workload.

When setting up CloudBB, the user can specify the number of SCBBs, i.e., choose the number of MNs.s As the user uses more SCBBs, CloudBB can handle more

metadata operations and become more scalable. Within a SCBB, actual data is stored on IONs, thus, as the user uses more IONs per SCBB, more I/O bandwidth can be provided. In other word, the user creates multiple PFSs on the fly depending on I/O intensity, CloudBB aggregates these PFSs and provide as a single scalable filesystem.

3.1.2 Hierarchical I/O Operations

CloudBB caches and buffers popular files on burst buffers so that CloudBB can accelerate I/O performance with respect to both I/O latency and throughput. In this section, we detail the hierarchical I/O operations in CloudBB (Figure 3.1).

Open

Once an SCBB is determined by the hash function for an open operation, the CN sends an open request to the MN in that SCBB (Op. 1 in Figure 3.1). When the MN receives the open request, the MN checks if the file exists in level-1 storage, i.e., SCBB. If the file is cached in level-1 storage, the MN sends the unique file id (`fid`) associated to that file, back to the CN (Op. 2). If the file does not exist in level-1 storage but exists in level-2 storage, the MN first assigns a new `fid` to that file and select multiple IONs within the same SCBB to store the file. The number of ION will be selected is determined by the number of replications set by users (More details in Section 3.1.3). If the open request is a create, i.e., the file does not exist in neither level-1 nor level-2 storage, the MN creates the file, assigns a new `fid` to the file and select multiple IONs for storing the file similar to the open case. We employ a simple round-robin scheme to select ION from ION set in the SCBB in order to distribute I/O workloads across IONs. `fid` is used for locating the distributed file chunks stored across IONs for successive I/O operations, such as read and write.

CloudBB supports a similar page cache scheme to the scheme in Linux kernel. To maximize I/O throughput by file caching, a file is divided into fixed size of *file chunks* (More details in Section 3.3.2).

Read and Write

When a CN reads a file, the CN sends a read request to the MN determined by the hash function (Op. 1). The read request includes `fid`, read size and the offset. Once the MN receives the read request, the MN finds the primary ION (More details in Section 3.1.3) storing the file with reference to metadata and sends a *chunk map* back

to the CN (Op. 2). The chunk map includes chunk location information, i.e., which file chunks belong to which ION. After receiving the chunk map, the CN directly connects to the corresponding ION (Op. 3), and then the ION sends requested file chunks to the CN (Op. 5). If the file does not exist on the ION but exist in level-2 storage, The ION first read the file chunks from level-2 storage before sending the file chunks (Op. 4).

A write operation is handled similarly to a read operation. In a write operation, however, the CN may append data to an existing file, i.e., `O_APPEND`. Because a file is divided into file chunks, additional file chunks needs to be allocated on the ION. When appending data to the file, the MN sends an allocation request to the same ION, appends pairs of a new chunk and an ION to the chunk map, and then sends the updated chunk map to the CN.

CloudBB also supports a metadata caching scheme. If the metadata is cached in the CN, the CN can omit communications to the MN. i.e. Op. 1 and Op. 2 (More details in Section 3.3.3). As mentioned, CloudBB supports a page cache scheme. If the file chunks are cached in the CN, the CN can omit communications to the ION, i.e., Op. 3 and Op. 5, and read and write operations are done via the local cache.

Flush and Close

When `flush` is called, the CN first checks if there are *dirty* chunks, i.e., chunks are updated on local cache but not synchronized with level-1 and level-2 storage. If the CN has dirty chunks in the local cache, the CN writes back to the level-1 storage. When a CN closes a file, the flush operation is invoked first, and then the CN sends a close request to the MN.

Other I/O Operations

CloudBB also manages file permission and ownership, and supports the other I/O operations such as `getattr`. CloudBB works as a common Linux filesystem by using `FUSE` (More details in Section 3.2).

3.1.3 Fault Tolerant Filesystem

Besides the performance, reliability is also an essential aspect for I/O subsystems. Since CloudBB caches data on IONs, cached data will be lost if any of IONs fails. To prevent such data loss caused by node failures, we also implement fault tolerance capabilities in CloudBB. CloudBB replicates data across different IONs and creates

metadata backup in reliable persistent storage (Section 3.1.3), and semi-automatically recovers from failures (Section 3.1.3).

Data Replication and Metadata Backup

As we mentioned, we buffer I/O data in the main memory of IONs. But, if cached files on level-1 are not written back to level-2 storage before node failures, the cached files will be lost. To prevent data loss, CloudBB creates multiple replicas across different IONs within the same SCBB, and elects one of the replicas as *primary* replica. We define the number of replicas as R, i.e., one for a primary replica of a file and $R-1$ for other replicas of the file. Depending on the reliability of clouds, the user can configure R.

When a CN creates a file, an MN allocates a primary ION and $R-1$ other IONs as secondaries for the file replication, and then the MN sends the information about the secondary IONs to the primary ION. When the CN writes or updates a file, the CN directly sends the data to the primary ION. After the primary ION locally receives the data, the primary ION synchronizes the file with other replicas on secondary IONs. We synchronize the replication using a different thread, so that the replication will not block the regular processes of IONs. In order to prevent data loss, the primary ION replies to CN after completing the synchronization. In read operations, MNs ensure that CNs read data from the primary replicas. Therefore, we can guarantee that CNs always read the latest version of files.

CloudBB also backups metadata. Since metadata size is much smaller than actual file size, synchronization between primary metadata and its backup is lightweight. Therefore, we simply backups metadata to a directory in reliable persistent storage specified by the user instead of creating multiple replicas of metadata across MNs.

Failure Detection, Notification and Recovery

In CloudBB, an MN and IONs periodically exchanges heartbeat messages within its SCBB. If either an MN or an ION does not receive any response for a certain period of time (*timeout*), CloudBB regards the quiet node as failed. If the failed node is one of the IONs, the MN elects one non-failed ION and restores files on the elected ION from replicas. The file replica recoveries are asynchronously done, and CNs can restart I/O operations before all of replicas are restored so that we can minimize the downtime. If the failed ION stores a primary copy of a file, MN promotes one of secondary copies to the new primary copy of the file.

On the other hand, failures of MNs are catastrophic. If metadata is damaged by MN's failures, manual recovery, e.g., rebooting a metadata server and running `fsck`, is important to ensure the consistency of the filesystem in practice. Fail-over for MN failures can cause more catastrophic secondary failures due to running applications under inconsistency in the filesystem. Although automatic recovery of MNs is as easy as one of IONs, we enforce the users to manually recover MNs to ensure the consistency. When an MN fails, CloudBB notifies the failure to the user so that the user can manually reboot the MN. Once the MN reboots, the MN starts recovering metadata from the backup first, and then also restore remaining metadata and check the consistency by collecting information (e.g. file size, chunk map), from IONs. After that, the user can manually run their own filesystem check processes before re-running applications.

In addition to heartbeat messages across an MN and IONs for failure detection, CNs can also detect failures on MNs and IONs if communications to either an MN or an ION fail. Therefore, once CNs detect failures, fail-over immediately starts for file recovery without waiting for *timeout*.

3.1.4 Highly Portable Filesystem Implementation

Providing standardized I/O interfaces is important so that existing applications can seamlessly run on CloudBB without modifying application codes. In order to intercept all the I/O operations from applications, we implement the client-side filesystem by using `FUSE` [36]. With `FUSE`, applications can access data in the same way as other existing filesystems (e.g. ext4), and existing applications can run on CloudBB without code modification. `FUSE` provides several I/O system calls to intercept I/O operations under mounted directories, such as open, read, write, flush and others. We modify these I/O system calls to redirect I/O requests to corresponding MNs and IONs by using socket communication with the TCP/IP protocol (Figure 3.1).

3.2 Implementation

In order to validate our proposal in real cloud system, we implemented our cloud burst buffer system with FUSE framework. In this section, we show the overview of the implementation (Section 3.2.1), describe the optimizations for performance (Section 3.2.2), and details of various file operations on our system (Section 3.2.7).

3.2.1 Implementation Overview

As described in previous sections, there are three kinds of nodes in our system: *Master Nodes*, *IOnodes*, *Compute Nodes*. Our implementation is based on the TCP/IP protocol and use sockets to communicate among *Master Nodes*, *Compute Nodes*, and *IOnodes*. We implement *Master Node* and *IOnodes* as servers and *Compute Nodes* as clients. To avoid slow down caused by hard disks, we buffer files in the main memory of each *IOnode*.

In order to intercept all the I/O operations from applications, we implement *Compute Nodes* with FUSE [36]. With FUSE framework, users can access data as other local files, and users' applications can work on our system without code modification. FUSE provides several hooks to intercept I/O operations under mounted directories, such as open, read, write, flush etc. We implement these hooks and redirect these I/O operations to corresponding *Master Nodes* and *Compute Nodes*.

Although FUSE supports multi-threaded operations, we only support single thread in the current implementation. As mentioned in Section 3.1.1 , although our proposed architecture, *Master Nodes* can control and balance the work load in each SCBB, in this paper, we focus on the effectiveness of CloudBB, so in current implementation, we use simple round-robin policy to assign file to *IOnodes* in each SCBB. In addition, we assume that all the files can be buffered in *IOnodes* and we don't perform write back. We also don't implement data replication for fault tolerance and data reliability in the current implementation. Although there are several ways to maintain the data consistency among burst buffer nodes when file data is replicated, currently the files are not replicated hence there is no eventual consistency problem described in Section 2 in our implementation. However these we mentioned above are the subjects of our future work to make our system more effective and robust.

3.2.2 Optimizations for Performance

FUSE makes it easy to intercept I/O operations and implement a userspace file system, nonetheless, targeted optimizations are required to achieve high performance. In this section, we introduce these optimizations in our implementation.

Firstly, we reduce I/O latency by reusing sockets. In TCP/IP protocol, creating a new socket needs three way handshakes and causes slow start. When each *IOnodes*

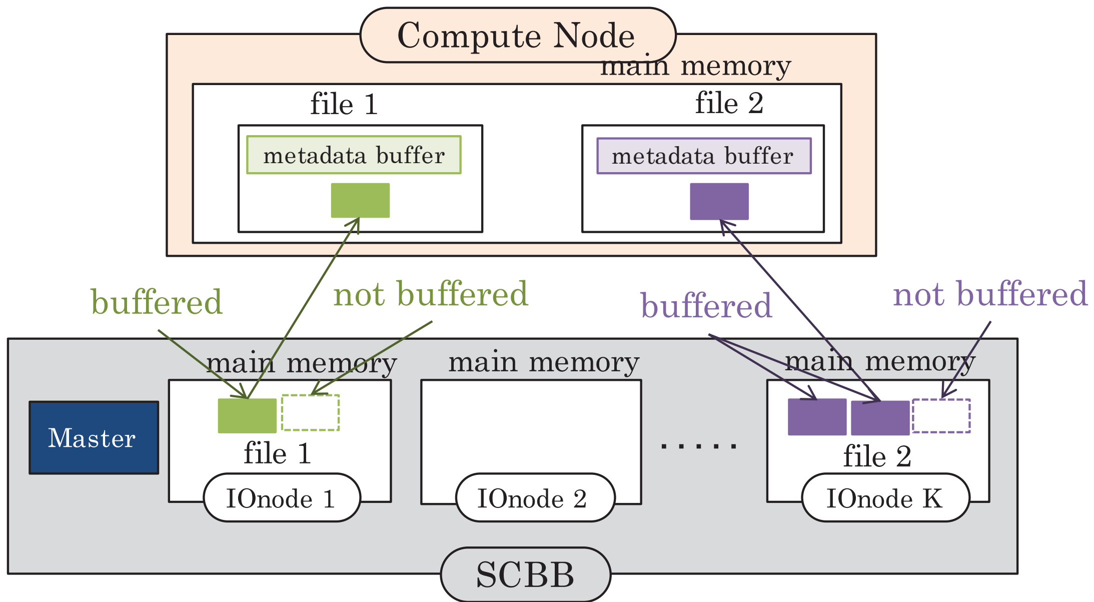


Figure 3.2: File Chunking and Buffers in Compute Nodes

starts up, it first registers itself to its *Master*, and when *Compute Nodes* start up they register themselves to all of the *Master Nodes*. They keep the sockets alive and reuse them for subsequent communications. *Compute Nodes* also register themselves to *IOnodes* at the first time they communicate to each other, not at the start up, so that *IOnodes* can be increased and decreased on demand. When *Compute Nodes* unmount burst buffer or *IOnodes* are shut down, they unregister themselves from *Master Node* and *Compute Nodes* also unregister themselves from all the registered *IOnodes*.

Secondly, we introduce data chunking into our system (Figure 3.2). Since we buffer data inside main memory of *IOnodes*, for some huge files whose size exceeds the size of main memory, we cannot buffer them entirely. Furthermore, due to the space limitation we may not be able to buffer all required file data inside memory. As we discussed in Section 2, data-intensive applications access data with high temportal I/O locality, and similar I/O locality exists in each task as is well-known, thus only a part of adjacent data in each file is required for processing within a short period for any timepoint. If we can buffer file data partially according to I/O locality and swap data in/out, we can effectively buffer only the data, which will be used in near future in limited memory regardless of the original size of the file. For these reasons, we split files into several fixed size data chunks and we allocate data chunk only when the chunk is requested. For example, in Figure 3.2 file 1 and 2 are split into 2

and 3 chuncks respectively according to their size, and the last chunks of both aren't buffered to save space. Additionally, we introduce dirty flags for each data chunk, which works like dirty flags in conventional memory management techniques. The flag is set to CLEAN when a chunk is first read from the remote shared storage, signifying that the buffered chunk is up to date. When the data chunk is updated, the flag changes to DIRTY and only DIRTY chunks need to be written back during the writeback phase; this reduces the unnecessary data transfer for write operations. Although distributing chunks of a single file over multiple *IOnodes* can achieve further performance and scalability, deciding the best chunk placement is difficult. In the current implementation, thus, we buffer all chunks of a single file are buffered in a single *IOnode*.

Thirdly, we introduce an additional tier of local memory I/O buffer in *Compute Nodes* for further performance improvement (Figure 3.2). In current FUSE (version 2.9.3) framework, FUSE framework ensures the I/O size of each request is not larger than one memory page size, which is 4KB in common Unix system. When users issue a request larger than memory page size, FUSE framework divides it into several sub requests, less than one page size each. Such implementation increases the number of requests greatly and thus reduce the I/O performance especially on remote file system such as our system. As mentioned in Section 3.1.1, although we adopt several optimizations to alleivate the overhead caused by large number of requests such as using hybrid Master-worker and Key-value architecture to distribute request workload, and preventing *Master Nodes* from involving in actual data transfer in read/write operations. However in our implementation since all the I/O requests go through *Master Nodes*, greatly increased I/O requests can still cause *Master Nodes* to become bottleneck. Furthermore, *Compute Nodes* need to communicate with *Master Nodes* and *IOnodes* before actual data transfer each time, increasing I/O requests will reduce the throughput for large size sequential I/O. Similarly, due to the I/O locality inside each process, if we buffer data locally inside *Compute Nodes*, subsequent access can be served locally. With such local buffer, we can reduce the number of I/O requests needed to be forwarded to *Master Nodes* and *IOnodes*, and hence improve the I/O performance. Hence we introduce an additional local memory buffer tier inside each *Compute Node*. These local memory buffers work like I/O buffers in common Unix system, we buffer I/O data in local memory and fulfill the subsequent requests access to the buffered range locally. For example, in Figure 3.2, we use the same size for local buffer in *Compute Nodes* as chunk size in *IOnodes* , and buffer the first and second data chunk of file 1 and 2 respectively

inside *Compute Node* to accelerate subsequent I/O operations. We also introduce a dirty flag for each local buffer to reduce the unnecessary data writeback, like dirty flag algothrim we introduced in data chunk, writeback only occurs when dirty flag is DIRTY. Actual data transfer between *IOnodes* only happens in following situations:

- Read, write or seek out of local buffer buffer range.
- FUSE framework calls flush on file with DIRTY dirty flag.
- FUSE framework calls close on file with DIRTY dirty flag.

Finally, we introduce local metadata buffers in each *Compute Nodes* (Figure 3.2). As we discussed in Section 2.4.2, besides the throughput, other file metadata operations can also affect the I/O performance, such as getattr in FUSE, which means get the file status. Since all of the *Master Nodes*, *IOnodes* and *Compute Nodes* are not multithreaded in current implementation, it will causes high latency and slow down other file operations if all these file metadata operations are forwarded to *Master Nodes*. To prevent this circumstance, we create local metadata buffers to cache file metadata in each *Compute Nodes* to accelerate the file metadata operations. However, we cannot buffer all the file metadata in order to keep consistency among all the *Compute Nodes*, because the cached metadata will become stale when other nodes update that file. Hence in our implementation, only the metadata of locally opened files are buffered. When a user opens a file, *Compute Node* retrieves file metadata from corresponding *Master Node*, and buffers it for subsequent file metadata operations, and when user applications update metadata, the operations will be performed on local metadata buffer and the buffer is marked as DIRTY. When flush operation is called for the file whose metadata buffer is marked as DIRTY, *Compute Node* will first update *Master Nodes*' metadata with locally buffered metadata, and then transfer data to *IOnodes*.

enditemize

3.2.3 Architecture and Internal Data Structure

3.2.4 Master

In this section we introduce the details in Master implementation. We first describe the architecture of Master progress to show how Master handles request from Clients and IOnodes, and manages IOnodes in the same SCBB. Then we show the details about internal data structure of Master, which supports all the operations.

Architecture

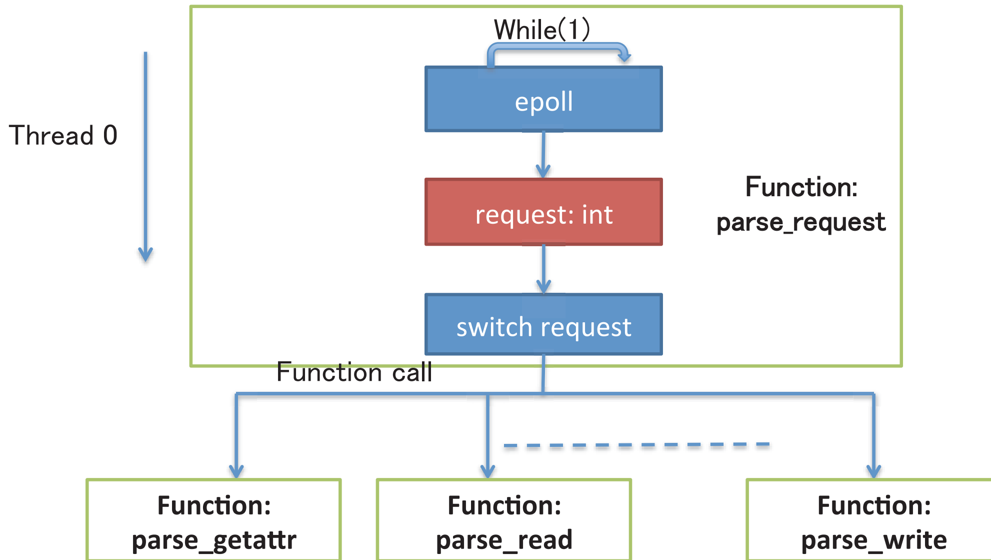


Figure 3.3: Master Architecture

Function	Description
parse_open_file	handle open file request from Clients
parse_read_file	handle read file request from Clients
parse_write_file	handle write file request from Clients
parse_flush_file	handle flush request from Clients
parse_close_file	handle close request from Clients
parse_attr	handle get file attribute request from Clients
parse_readdir	handle read directory request from Clients
parse_rmdir	handle remove directory request from Clients
parse_unlink	handle remove file request from Clients
parse_access	handle
parse_rename	handle rename request from Clients
parse_truncate	handle truncate request from Clients
parse_new_client	register new client socket
parse_register_IOnode	register new IOnode socket
parse_close_socket	close a registered socket

Table 3.1: Master Functions

As we mentioned in Section 3.1, Masters are the supervisors in each SCBB, and serve as metadata server and control all the IOnodes in the same SCBB. We implemented Masters as servers, the architecture of Master process is shown in Figure 3.3.

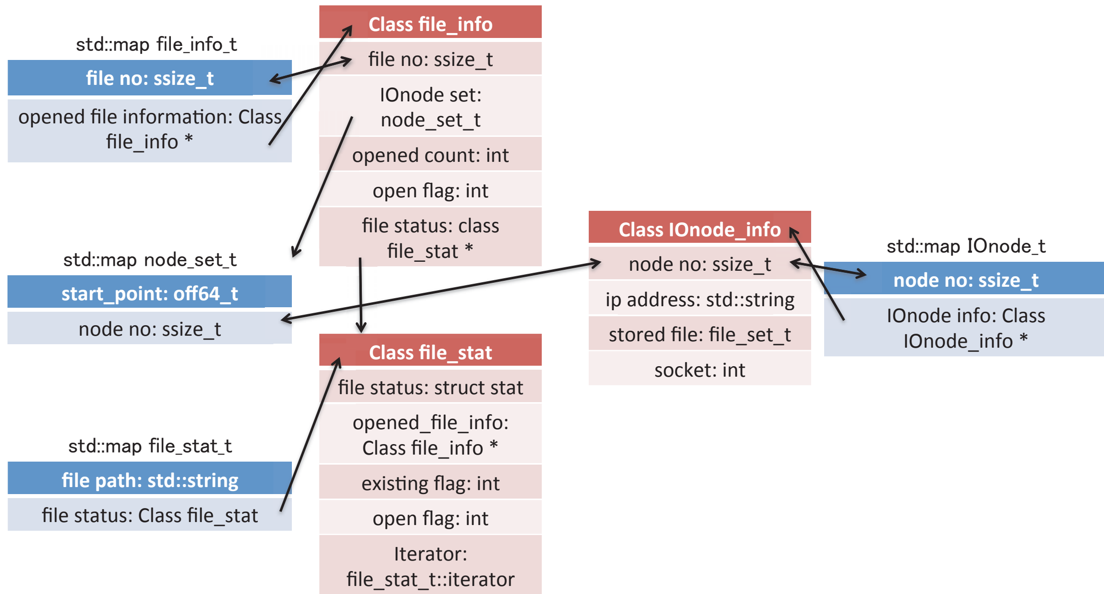


Figure 3.4: Data Structure in Master

First we describe the main progress of Master. When the Master progress starts, it initializes several internal data structures, and starts to listen on a specific port. The port can be specified through configure file. As When each IOnode starts, they first connect to corresponding Master, and register this socket to Master. Furthermore, in our implementation, we reuse socket to reduce the cost for creating a new socket for each time connection. These mean Master need to wait on multiple sockets, and handles request from these sockets. In current implementation, as shown in Figure 3.3, we use epoll to wait on multiple sockets.

When a new request comes, kernel notifies Master progress with corresponding socket through epoll. Then Master receives a integer number represents request number, and call corresponding handler to handle this new request. We define a function to handle each request, Table 3.1 shows all the function names and the request they handle. Most functions handle requests from FUSE framework forwarded by Clients, "parse_new_client" and "parse_register_IOnode" are responsible for registering new client and IOnode socket for epoll, and "parse_close_socket" does the reverse operation, close a registered socket.

Data Structure

Here we describe internal data structure in Master. In order to handle the request described in Table 3.1, Master maintains following two kinds of data structures:

- File metadata, which stores file metadata informations including file size and file permission.
- IOnode information, which stores the informations about registered IOnodes.

Figure 3.4 shows the details about internal data structure. For each file we assign a unique file no when Clients first open it, and Clients use this file no to perform subsequential I/O operations on that file. Likes file no, we also assign a unique node no for each registered IOnode. Since Master needs to manage a huge mount of files as well as many registered IOnodes, and in each I/O request, query for file or IOnode information via file no and node no occurs frequently, in order to accelerate these query performance, We use STL map to store these information. Since STL map was implemented as Red-Black tree, a kind of balanced binary search tree, it costs $\mathcal{O}(\log n)$ for insert, delete and search. Hence we can still achieve good performance with such STL map even buffer a huge mount of files. As shown in figure 3.4, we set file no or node no as key, and class which store the informations as value. There are three classes with four maps:

- **Class file_stat**
  Class file_stat stores the basic file status of each buffered file. We insert file_stat objects into std::map file_stat_t as value, and the key is file relative path.
  - *file_status* is a linux standard file status structure, struct stat, to store the file status of buffered file.
  - *opened_file_info* is a pointer to a Class file_info object, which stores the some additional opened file informations, the pointer is NULL if the file hasn't been opened by any clients.
  - *Existing flag* is set to true when the file is already existed on remote storage. When client creates a new file, we create a new file_stat object and the corresponding existing flag is set to false instead of creating a new file on remote storage. Such lazy creation strategy helps to reduce the latency of file creation.
  - *iterator* is an iterator of std::map file_stat_t. We insert file_stat objects into file_stat_t as value, and key is file relative path, we store the iterator of each file_stat objects in the last field so that we can get the file relative path, which is the key field from that file_stat.
- **Class file_info**

Class file_info stores some additional opened file informations. We insert file_info objects into std::map file_info_t as value, and the key is file_no.

- *file_no* is the unique file id we assign to each opened file.
- *IOnode_set* maintains a block-start-point/IOnode-id map. Recalling that we divide each file into several fixed size blocks (Section3.2.2), here block-start-point represents the start offset of each block.
- *open_count* maintains a open counter on each file, counter increases when the file is opened by clients, and decreases when clients close that file.
- *file_status* maintains a pointer to Class file_stat object, which stores the basic file status of the corresponding files.

• **Class IOnode_info**
Class IOnode_info stores information about each IOnode. We insert IOnode_info objects into std::map IOnode_t as value, and the key is IOnode_no.

- *IOnode_no* is the unique id we assign to each IOnode.
- *ip_address* is the ip address associated with the IOnode.
- *socket* stores the socket associated with the IOnode, we keep all the socket alive.

3.2.5 IOnode

Subsequently, in this section we introduce the details in IOnode implementation. We first describe the architecture of IOnode progress to show how IOnode handles request from Clients and Master, manages buffered file blocks, and transfers data with Clients. Then we show the details about internal data structure of IOnode, which supports all the operations.

Architecture

Recalling that IOnodes are under the control of corresponding Master, store actual I/O data in fixed blocks, and are responsible for transferring data with Clients. We implemented IOnodes as both as servers and clients, the architecture of IOnodes process is shown in Figure 3.5.

Then we describe the main progress of IOnode. When the IOnode progress starts, it initializes several internal data structures, starts to listen on a specific port,

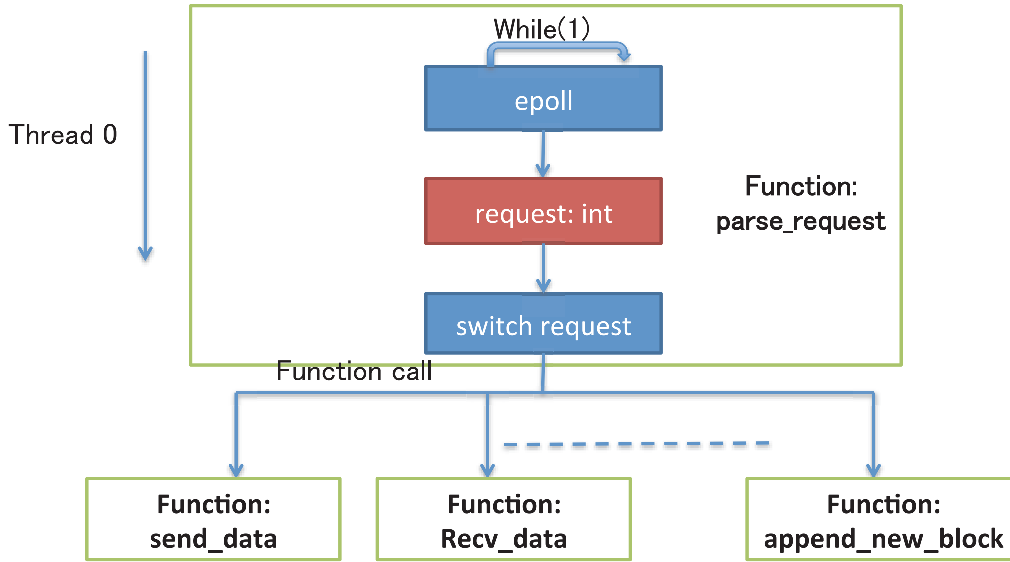


Figure 3.5: IOnode Architecture

Function	Description
send_data	send data to Clients
recv_data	receive data from Clients
open_file	handle the open file request from Master, register a new file
close_file	handle close request from Master, remove file
flush_file	handle flush request from Master
rename_file	handle rename file request from Master
truncate_file	handle truncate file request from Master
append_new_block	append a new block to an existing file
parse_new_client	register new client socket
parse_close_client	close a registered client socket

Table 3.2: IOnode Functions

and register itself to corresponding Master. The IP address of Master is configured through environment value and the listening port can be specified through configure file. Like Masters, we reuse socket to reduce the cost for creating a new socket for each time connection, so IOnodes also use epoll to wait on multiple sockets.

Like Masters, we assign a unique integer number to identify request, IOnodes use this number to call corresponding handler to handle this new request. Table 3.1 shows all the function names and the request they handle. Most functions handle requests from Master, perform operations on buffered files. Masters send "append_new_block" request to corresponding IOnode when Clients write over the end of file, IOnode append a new block and then receive data from Clients.

"parse_new_client" is used to registering new client sockets for epoll, and "parse_close_client" close the corresponding socket.

Data Structure

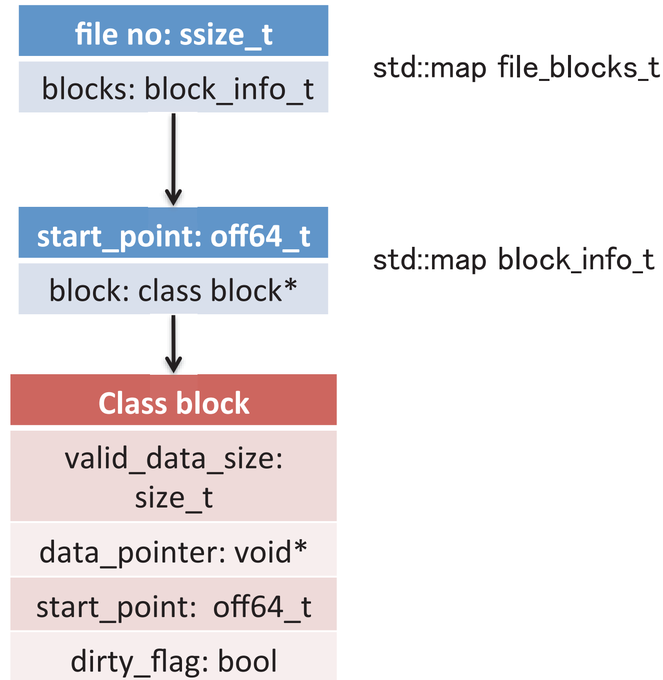


Figure 3.6: Data Structure in IOnode

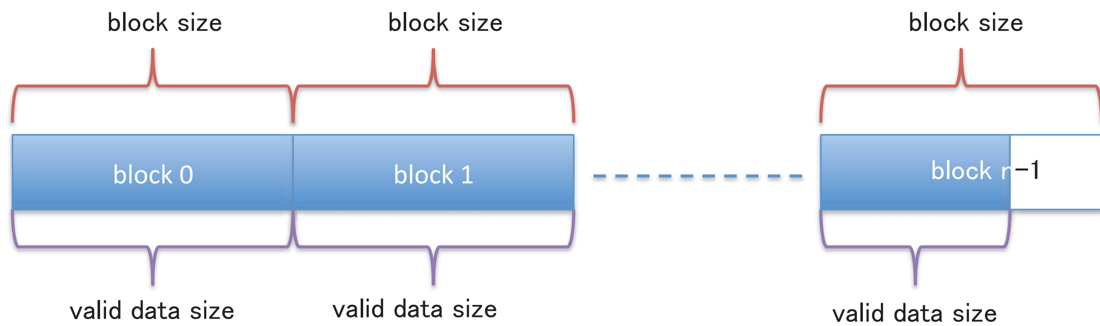


Figure 3.7: Block Size and Valid Data Size

Then we describe internal data structure in IOnode. In order to handle the request described in Table 3.2 Figure 3.4 shows the details about internal data structure. There are a class and two maps:

- **Class block**

Class block stores the most basic informations and actual data for a block of each buffered file. We insert Class block objects into std::map block_info_t as value, and the key is block start offset, then insert all the blocks belongs to the same file into std::map file_block_t.

- *data_size* maintains the valid data size in the block, Figure 3.7 shows the relationship between fixed block size and valid data size.
- *data* is a pointer to the actual data block. Here we adopt lazy memory allocation policy, which means we allocate memory only when the block is read or written for the first time.
- *start_point* stores the start point of the block.
- *dirty_flag* is set to clean as default value, and set to dirty when there are any modifications on that block. We only write back the dirty block to reduce the unnecessary write back data transfer.

3.2.6 Client

Finally, in this section we introduce the details in Clients. As usual, we first describe the architecture of Client progress. Then we show the details about internal data structures of Clients.

Architecture

As we described in previous section, we implements Clients with FUSE framework, the architecture of FUSE framework is shown in Figure 2.18. Figure 3.8 shows the relationship of each layer in Client architecture, and the data flow of each I/O function calls. Clients register several hooks (Table 2.3) in FUSE framework when they set up. When users applications access the mount directory, FUSE framework handles these requests and call the corresponding registered hooks, then clients forward I/O requests from FUSE framework, and communicate with Masters and IOnodes to fulfill the requests.

We implemented three layers inside our client process (Figure 3.9):

- Low-level POSIX layer
  This is the lowest layer in our client process. This layer handles all the communications with Masters and IOnodes, forwords I/O requests from higher layer to Masters and IOnodes. As its name suggests, this layer exports low-level POSIX APIs like open, read, write etc..

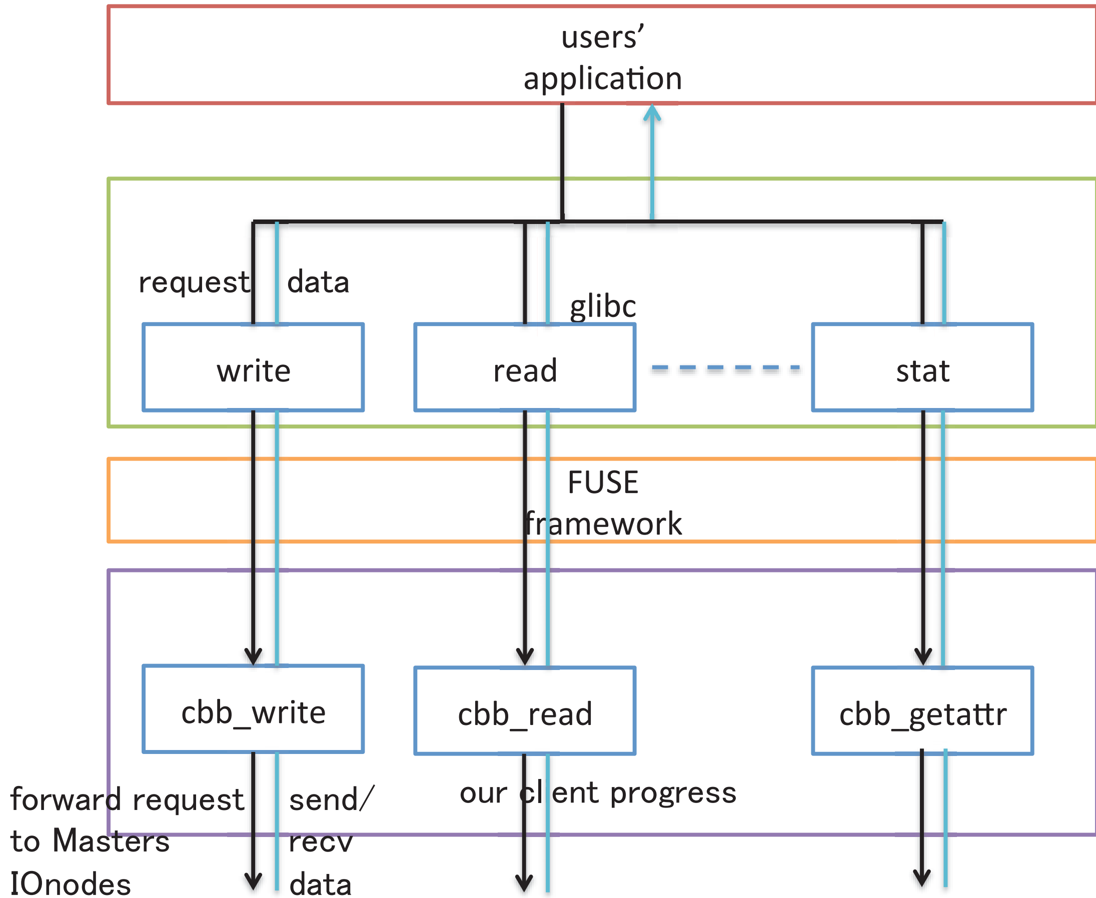


Figure 3.8: Client Architecture

- Buffered I/O layer
  As we mentioned in Section 3.2.2, we implemented local in-memory I/O buffer for each file to improve the I/O performance. The local in-memory I/O buffers are located in this layer. This layer exports standard buffered I/O APIs like fopen, fread, fwrite etc.. I/O requests from higher layer are served by I/O buffers, when the data isn't in I/O buffers or I/O buffers are full, the I/O requests are forwarded to Low-level POSIX layer.

- FUSE Hooks layer
  As its name suggests, this layer implements all FUSE hooks using APIs exported by Buffered I/O layer. When the client process start up, it registers all the hooks in this layers to FUSE framework.

We implemented low-level POSIX layer and buffered I/O layer as dynamic linked library. Hence all these three layers can be used separately, however higher layer

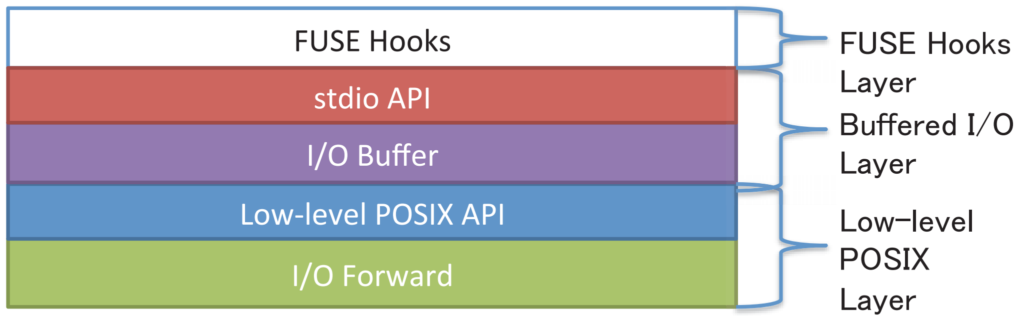


Figure 3.9: Client Process Detail

requires support from lower layer. For example, user applications can call our low-level POSIX layer APIs directly by pre-load our library before execution.

Data Structure

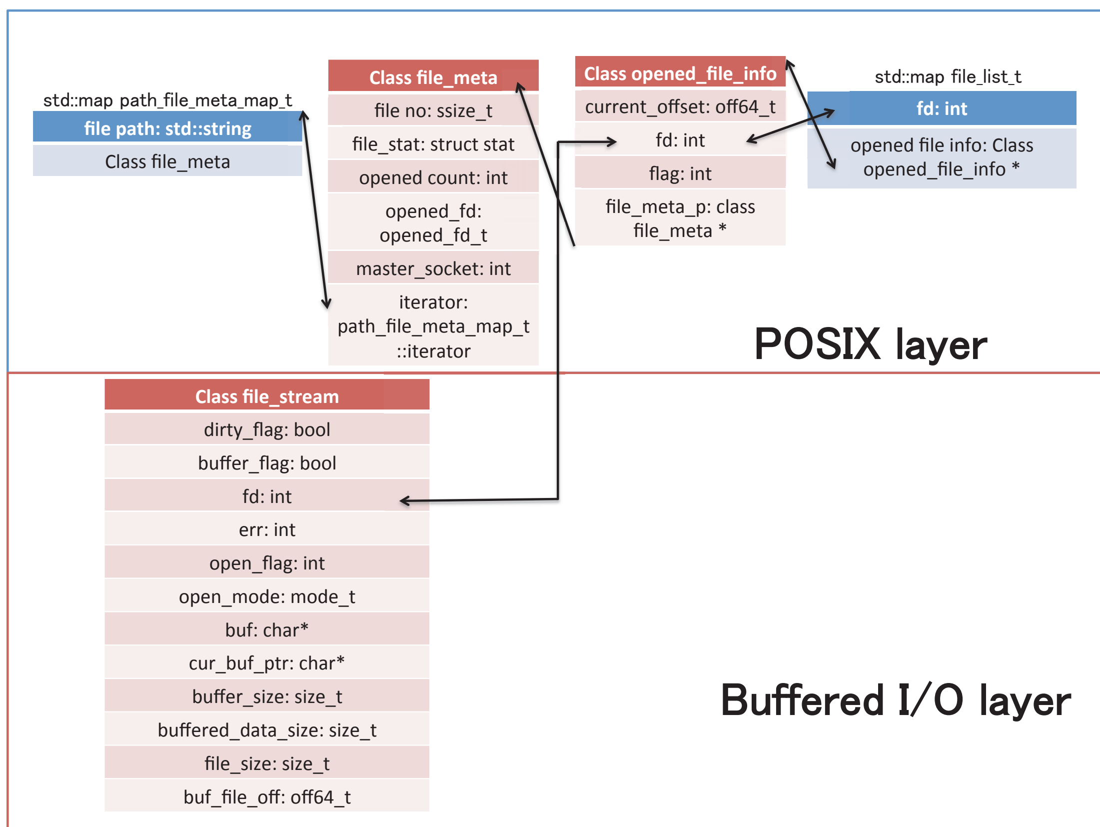


Figure 3.10: Data Structure in Client

Here we describe internal data structure in Clients. As shown in figure 3.10, there are two major layers as we introduced: POSIX and Buffered I/O layer.

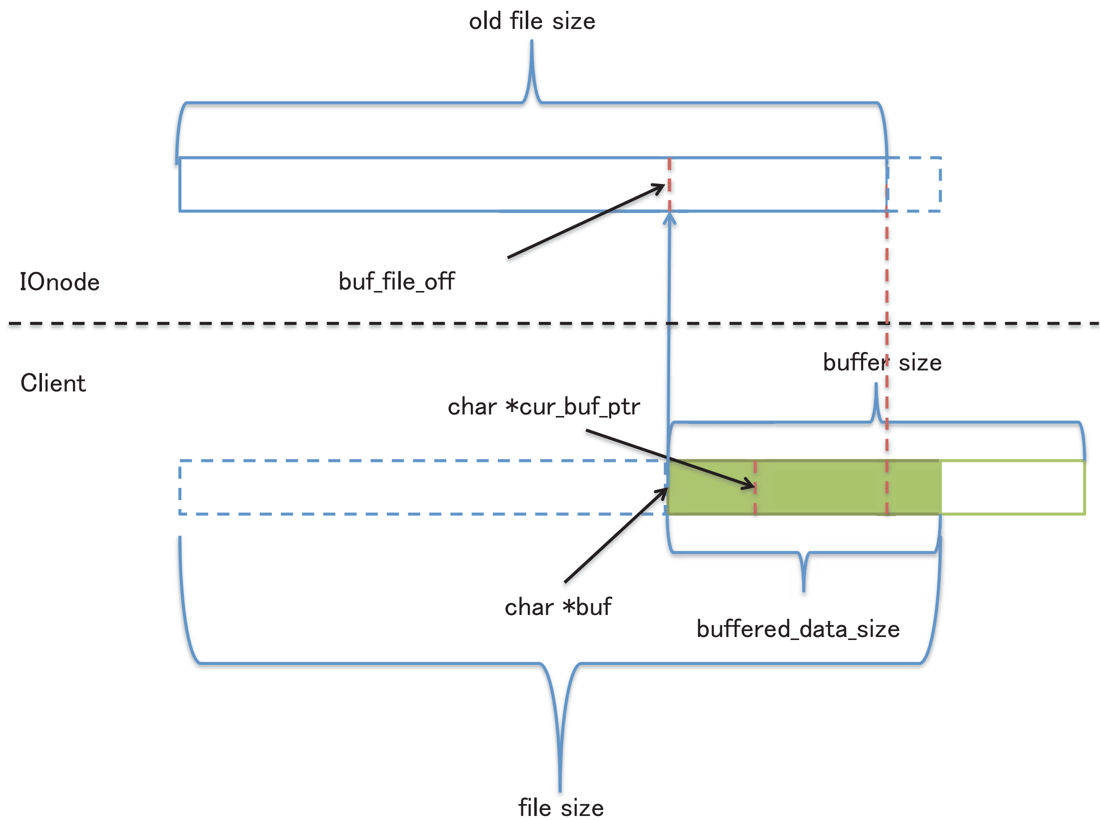


Figure 3.11: Client Buffer

- POSIX layer:
  There are two classes and two maps in this layer:
  - **Class file_meta**
    Class file_meta stores the basic meta data of each opened file. We insert Class file_meta objects into std::path_file_meta_map_t as value and the key is file relative path.
    * *file no* stores the file no returned from Masters, Clients use this file no to identify file with Masters and IOnodes.
    * *file_stat* stores the standard Linux/Unix file status information.
    * *opened count* maintains a counter on how many times the file has been opened.
    * *opened_fd* stores a list of opened fd associated to this file.
    * *master_socket* stores the socket which used to connect with corresponding Master, the value is determined when the file metadata is buffered.

    - *iterator* point back to the container. The file path can be retrieved from this iterator.

- **Class opened_file_info**

  Class opened_file_info stores the additional information associated to opened file. We create a opened_file_info object for each opened file. In order to use the POSIX layer separately, We assign each opened_file_info object a unique local fd, and return this fd to users. Users can use these fds to perform operators on file as standard Linux file descriptors. We insert pointer to Class opened_file_info objects into std::file_list_t as value and fd as key.

    - *current_offset* stores the current I/O offset.
    - *fd* stores the associated fd.
    - *flag* stores the open flag.
    - *file_meta_p* stores a pointer to associated file meta data.

- Buffered I/O layer:

  There is one class in this layer:

  - **Class file_stream** Class file_stream stores additional information used for local I/O buffer. Class file_stream maintains file stream can be used like standard Linux/Unix file stream. We create a Buffered I/O layer object for each opened file stream.

    - *dirty flag* maintains dirty flag associated to the file stream. As we introduced previously, only the stream with DIRTY flag will be written back.
    - *buffer_flag* indicates whether the stream associates with local I/O buffer.
    - *fd* stores the fd associated with the stream, we use this fd to forward I/O requests to POSIX layer.
    - *err* stores the latest err number.
    - *open_flag* stores the open flag.
    - *open_mode* stores the open mode.
    - *buf* is a pointer to the start address of local memory I/O buffer. The pointer is set to NULL initially or When buffer_flag is set to FALSE. The relationship of *buf*, *cur_buf_ptr*, *buffer_size*, *buffered_data_size*, *file_size*, *buf_file_off* is shown in Figure 3.11.

* *cur_buf_ptr* is a pointer to current I/O offset on the I/O buffer.
* *buffer_size* maintains the size of I/O buffer.
* *buffered_data_size* maintains the valid data size in the I/O buffer.
* *file_size* maintains the valid file size. Since we buffer the I/O data locally, when users write over the end of file, we update this *file_size* to the new size.
* *buf_file_off* maintains the file offset of I/O buffer.

3.2.7 File Operations

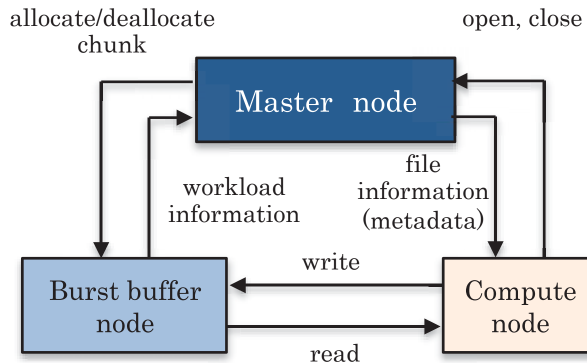


Figure 3.12: Dataflow of File Operation

In this section, we give more details about the file operations in our implementation: open, create, flush, close, read, write and other operations in FUSE. (Figure 3.12).

Open and Create

First, we introduce the open and create process (Figure 3.13). When an application opens a file, the FUSE framework calls our open function, and pass the relative file path from the mount point, and other information like file access mode. We first apply consistent hash against the file relative path to select the corresponding *Master Node*, and record this information in local data structure, then send the file path and file open flag to that *Master Node*, when *Master Node* receives the path, it first searches whether the file has been buffered already, If the file is already buffered, *Master Node* replies to the client with a unique id which will be used in the subsequent file operations. If the file hasn't been buffered, *Master Node* will first pick up a new file id, and select several *IOnodes* in the same SCBB, and send open request with the

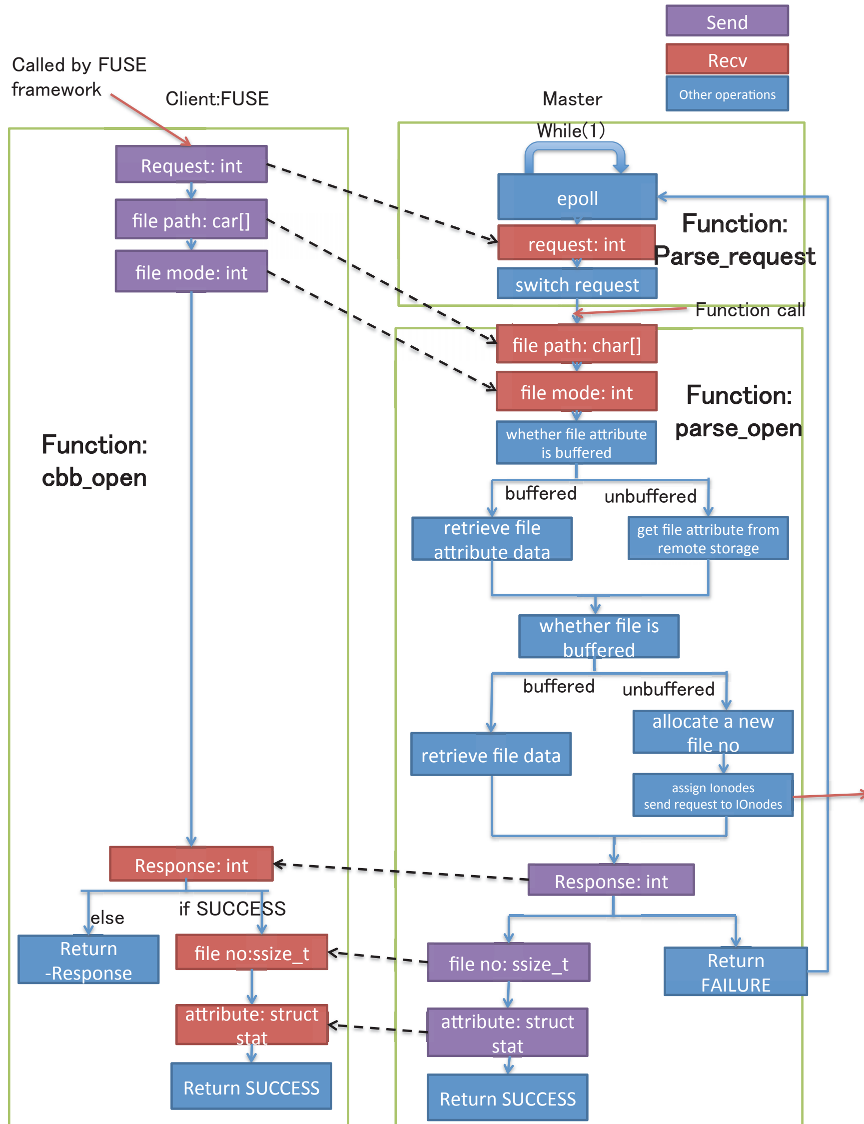


Figure 3.13: Process of open/create file

file relative path to these *IOnodes*, subsequently, *Master Node* returns the file id to *Compute Node*. *Compute Node* stores the file id in FUSE file meta-data and uses it for the following operations.

File creation is quite similar with file open, when *Compute Node* connects to *Master Node*, beside the file relative path and open flag, we also send the file mode, which denotes the permission in common Unix system. In order to accelerate file creation, *Master Node* creates a new file with the file mode, assigns some *IOnodes*, and return the file id. In our implementation *Master Node* don't actually create a

new file in remote shared storage, rather, *Master Node* marks the new created file with a not existing flag, and create file in remote shared storage when write back.

Read and Write

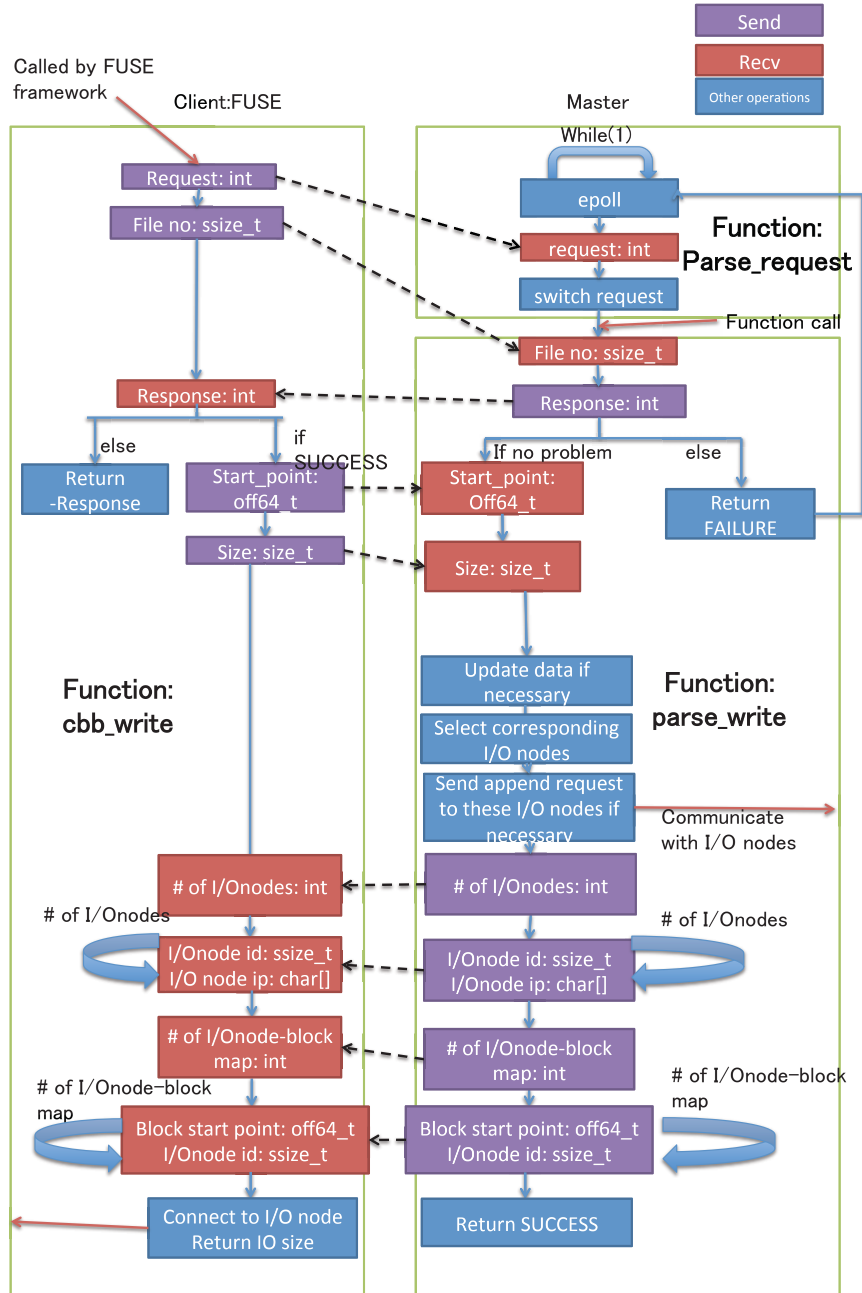


Figure 3.14: Process of Write

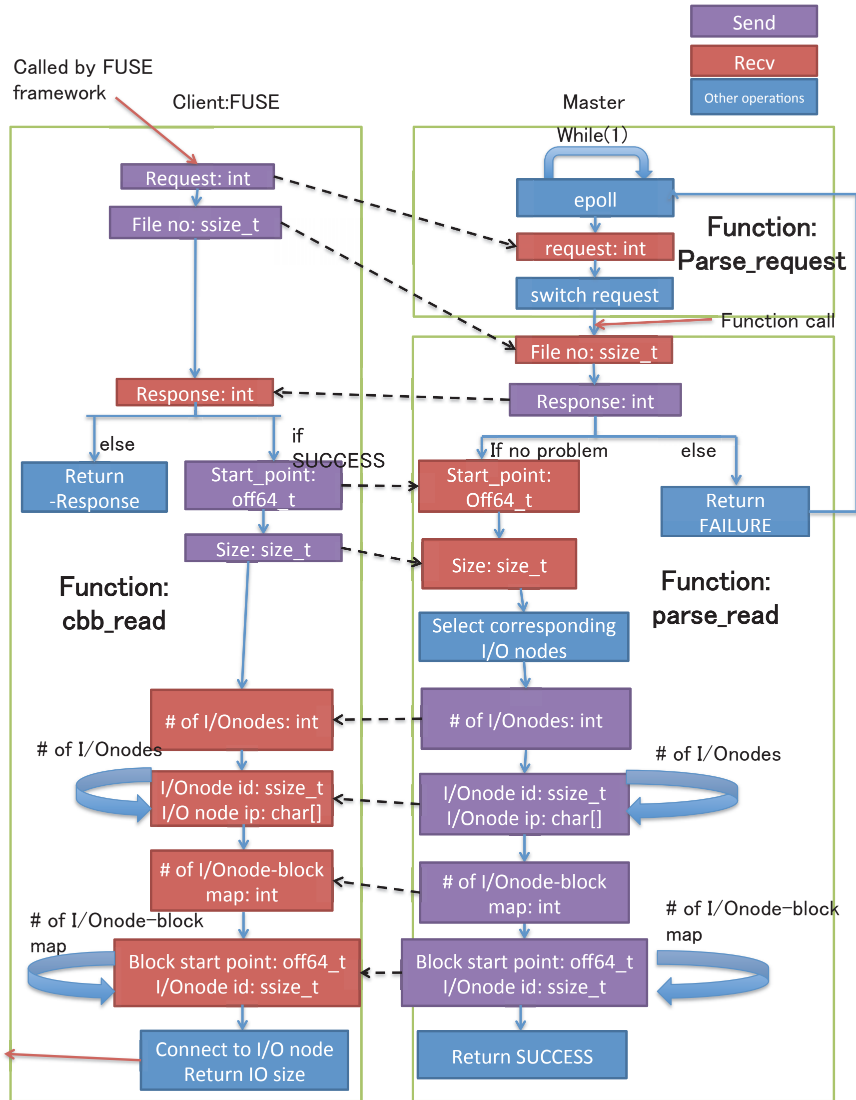


Figure 3.15: Process of Read

Next, we introduce the read and write process (Figure 3.14 and 3.15). When an application issues a read request, the FUSE framework calls our read function with file id, data buffer pointer, read size and offset. We first get the corresponding *Master Node* from local record, and then send the read request with read size and offset. As mentioned in section 3.2.2, files are split into several fixed-size data chunks, after receiving these information, *Master Node* responds to the *Compute Node* with a map of file chunk-*Onodes*, provides *Compute Node* with the information about which *IOnode* is responsible for each file chunk. After receiving the map, *Compute Node* connect to the corresponding *IOnode* directly, and sends the read request with size and offset. If the file is buffered in the *IOnode*, the *IOnode* will send *Compute Node*

with the requested data, otherwise, the *IOnode* will first read from remote shared storage, can then send the data.

Write operation is treated similarly to the read operation. In addition to the read case, we also have to consider the case that data is written beyond the current end of file. Since files are split into several fixed-size data chunks, data chunk needs to be additionally allocated in this case. When *Master Node* finds that data chunk allocation is required to fulfill write request by a *Compute Node*, *Master Node* will send the allocation request to the same *IOnode* which buffers the existing data chunks of the same file, as one file will be buffered in only one *IOnode* in current implementation. Subsequently *Master Node* appends this new chunk-*IOnode* pair into its chunk-*IOnode* map, and respond to the *Compute Node* with the new chunk-*IOnode* map.

Flush and Close

Figure 3.16 and 3.17 show the process of flush and close. As mentioned in section 3.2.2, in order to reduce the read/write operations between *Compute Nodes* and *masters*, *IOnodes* as well as the unnecessary write back data transfer, we introduce the local I/O buffer, and dirty flag. Hence when flush is called, *Compute Node* first checks the dirty flag of local I/O buffer. If it is set to DIRTY, *Compute Nodes* will call write on local I/O buffer, and set the dirty flag to CLEAN.

When user closes the file, the flush operation will be invoked by FUSE framework first, and then the close request will be sent to *Master Node* to perform actual file close.

Other Operations

In FUSE framework, other than the above mentioned operations, there are also several file meta data operations available such as getattr, which means get file status. We also implemented this and other file meta data operations in order to make our system behave correctly. Figure 3.18, 3.19, 3.20 show the details in each operation.

Besides these operations, since we use static hash of file path to determine the corresponding SCBB, operations like rename and symblink can change the map between file path hash value and the actual SCBB. For example, initially, file A belongs to SCBB A according to static file path hash. Then file A is renamed to file B, which belongs to SCBB B according to static file path hash, however the actual data remains in SCBB A. Similarly, file C in SCBB C can be a symblink to file A in

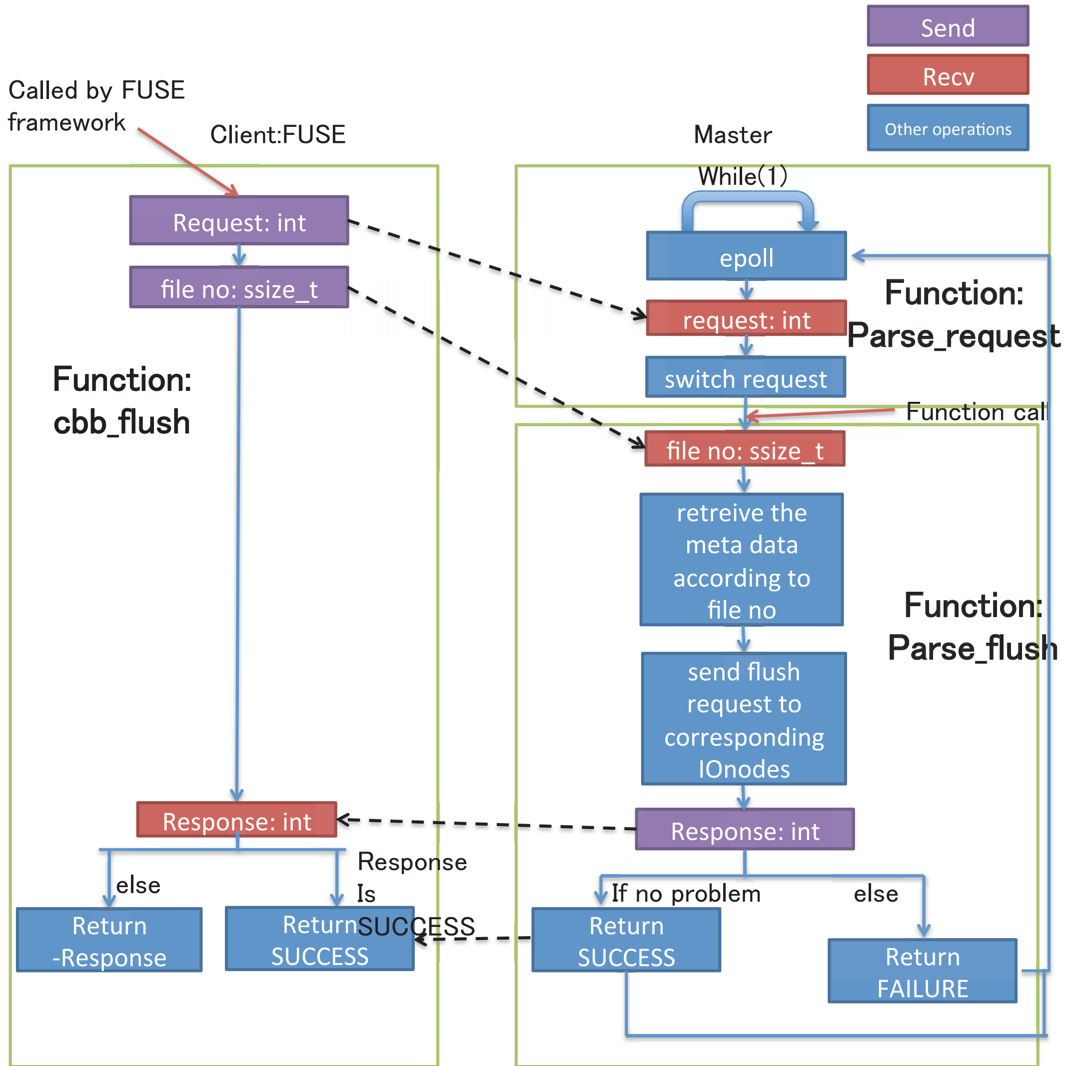


Figure 3.16: Process of Flush

SCBB A. In such situations, without file migration, connecting to SCBB B and C according to new file path or symblink can not get the actual data which remains in SCBB A. However file migration can cause unnecessary network transfer between IOnodes, and cause performance degradation. On the other hand, new file name after rename or symblink may locate in the same SCBB, cause no trouble in file access. In order to reduce the overhead of file migration caused by file rename and symblink creation in our implementation, we create a special file for the symblink file and renamed file in new SCBB. Inside of these special files, there is a pointer point to the original SCBB, when users access these files, Masters return the original SCBB name to users to redirect them to the original SCBB.

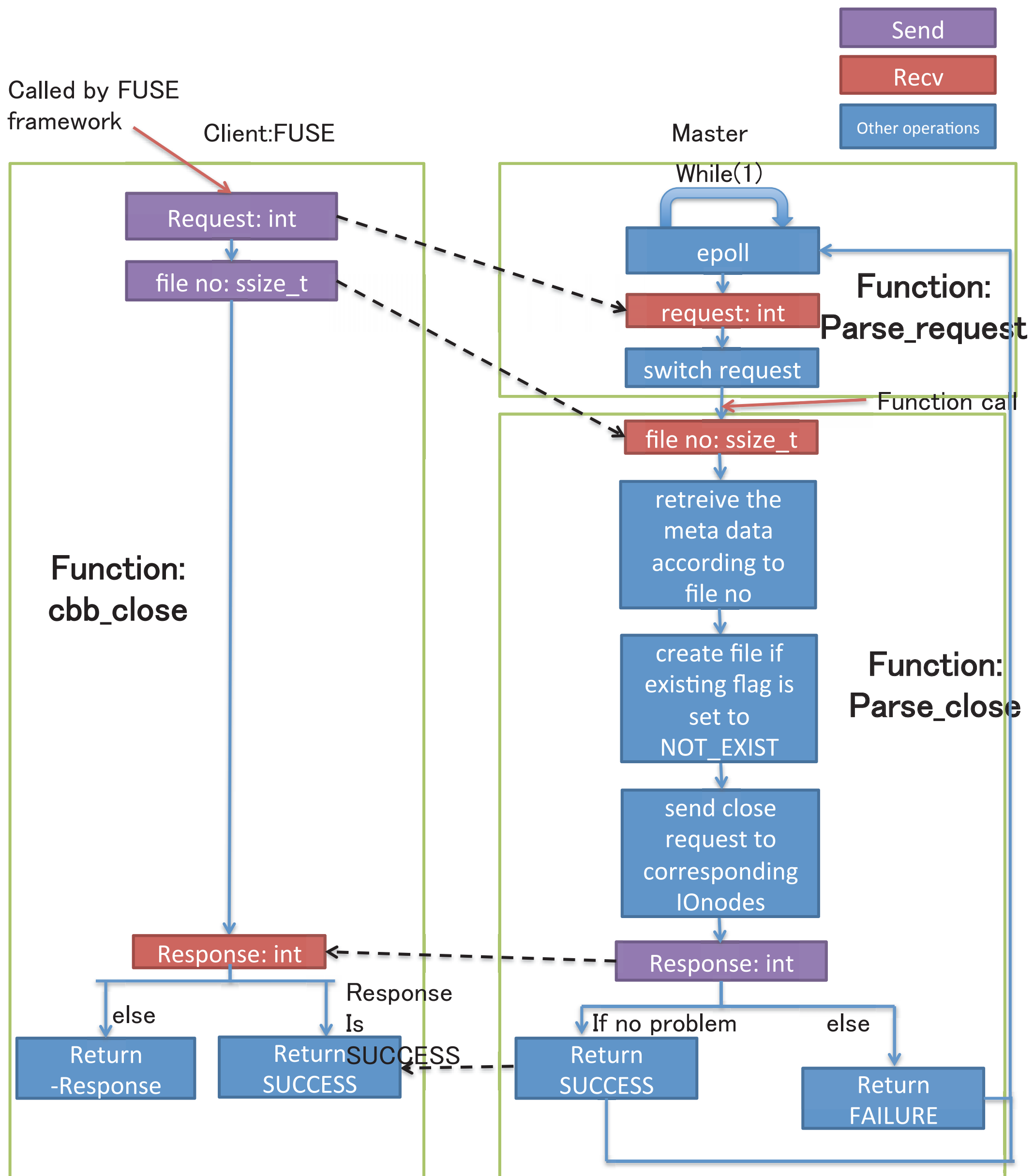


Figure 3.17: Process of Close

3.3 Optimizations

With FUSE, developers can easily intercept I/O operations and implement a filesystem in user space. However, targeted optimizations are required to achieve reasonably high performance. In this section, we introduce these optimizations in our implementation.

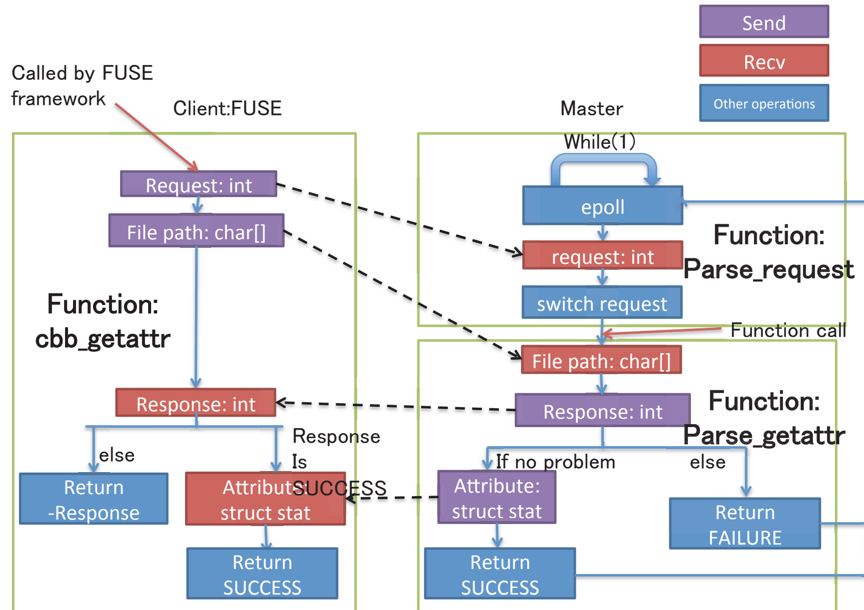


Figure 3.18: Process of gettattr

3.3.1 Socket Reuse

Firstly, we reduce I/O latency by reusing sockets. In TCP/IP protocol, creating a new socket requires three way handshakes and causes slow start. When each ION starts up, each ION first registers itself to its MN. When CNs start up, the CNs register themselves to all of the MNs. MNs, IONs and CNs keep the sockets alive and reuse them for subsequent communications and failure detection. CNs also register themselves to IONs at the first time when CNs and IONs communicate each other. When CNs unmount CloudBB or IONs are shut down, CNs and IONs unregister themselves from MNs, and CNs also unregister themselves from all the registered IONs.

3.3.2 File Caching

The latest `FUSE` (version 2.9.3) framework limits size of each write and read operations to no larger than memory page size, typically 4 KB. When users issue a write or read request whose size is larger than the memory page size, `FUSE` divides the I/O request size into multiple small I/O requests whose sizes are no larger than the memory page size. Such implementation increases the number of I/O requests in large file I/O, thereby, degrades the I/O performance because CNs have to communication with MNs whenever I/O requests are issued.

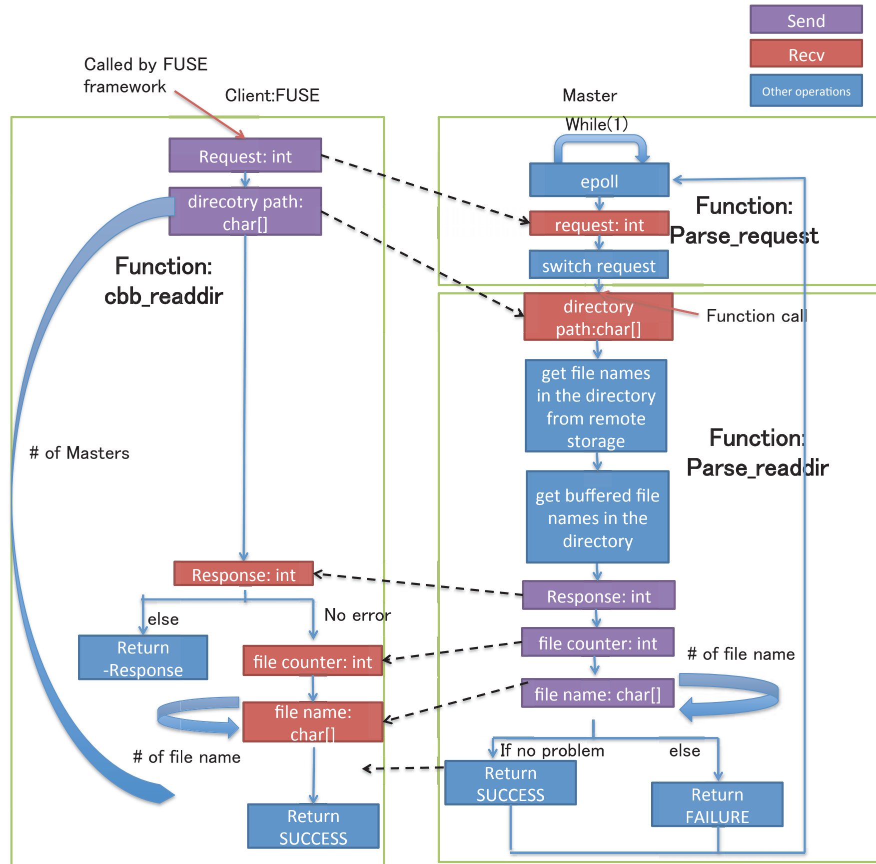


Figure 3.19: Process of readdir

To avoid such small communication with MNs, CloudBB supports a page caching scheme in CNs in similar algorithm to the Linux kernel. We buffer file chunks in local memory of CNs and handles subsequent I/O requests via the local cache without communicating with MNs and IONs. For the consistency between the local cache and level-1 storage, we use dirty flags for each file chunks similar to the page write-back algorithm in the Linux kernel. CloudBB write-backs dirty file chunks to level-1 storage when the CN reads, writes or seeks out of local buffer range, or calls flush/close. After file close, data in local cache could be stale if other CNs update the same file. To avoid accessing on stale data, we remove the local cache right after the file close, and retrieve the latest data from IONs again in reopen to guarantee the consistency.

3.3.3 Metadata Caching

In CloudBB, CNs also cache metadata to accelerate metadata operations. We create local metadata buffers to cache file metadata in each CN. When a CN opens a file,

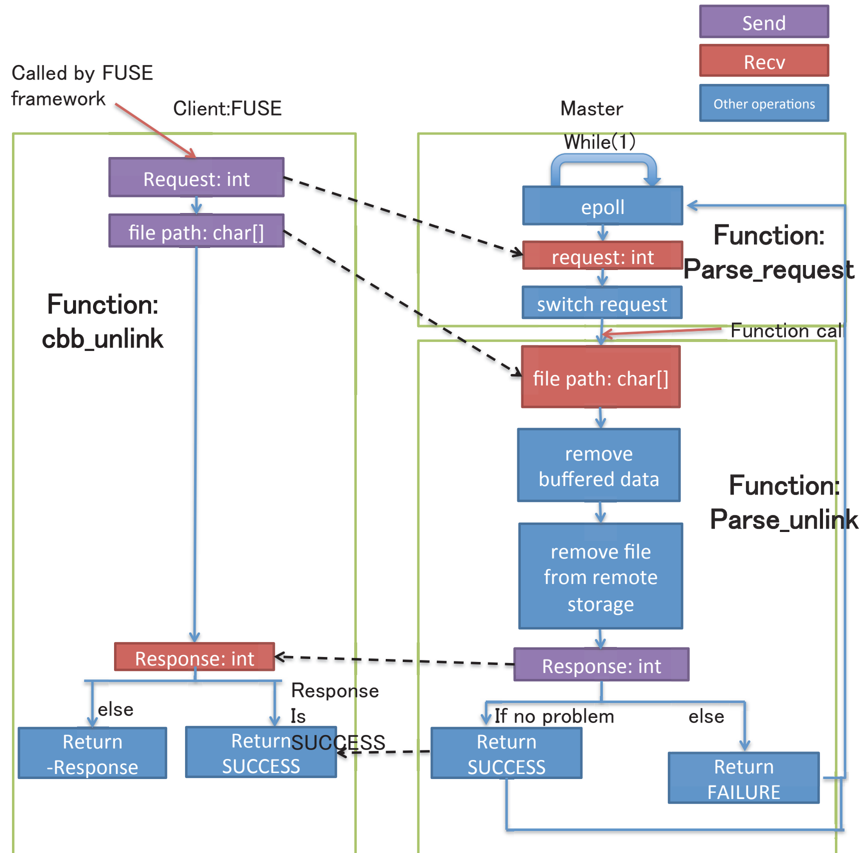


Figure 3.20: Process of unlink

the CN retrieves file metadata from the corresponding MN, and caches the metadata. When the CN updates metadata, the operations are performed on locally cached metadata, and then the metadata is marked as dirty. When flush or close is called on the file whose metadata is marked as dirty, the CN synchronize locally cached metadata with MNs' metadata, and then delete local metadata cache.

3.4 Evaluation

To validate the effectiveness of CloudBB, we conduct several evaluations on Amazon EC2/S3 in the Tokyo region. For CNs, IONs and MNs, we use the same instance type shown in Table 3.3. Here vCPUs means the number of virtual CPUs provided to instances. For the price, since the price of a spot instance is based on auction and changes all the time, here we use the price at 2016/05/12 12:00 UTC+9 when computing total cost for spot instances. We mount Amazon S3 via `s3fs` [18] for shared cloud storage, i.e., level-2 storage, and CNs mount CloudBB via our FUSE

Table 3.3: Evaluation environment

Cloud provider	Amazon EC2
Region	Tokyo
Instance type	m3.xlarge (spot instance)
vCPUs	4
Memory size	15GiB
Network performance	High
Price	$0.045/hour (2016/05/12 12:00:00 UTC+9 ap-northeast-1c)

filesystem. We set the *file chunk* size to 5 MB and local file I/O buffer size to 100 MB. Since 3-way replication is the most common redundancy in data centers, which guarantees a good balance between performance and reliability [6, 8, 38]. We create three copies for each file in the all evaluations, i.e., $R = 3$. Thus, we use three IONs per SCBB at the minimum. We configure MNs to send heart beat check to all the ION every 1000s to avoid huge overhead caused by frequent heart beat check.

3.4.1 Comprehensive I/O Pattern Evaluations

Sequential and Random I/O

To illustrate how our system solves the I/O performance issue, we first evaluate sequential and random I/O performance with a single SCBB, i.e., the number of MNs is one. We use IOR [72] to measure the sequential and random performance. From the nature of the ClouddBB architecture, CloudBB can concurrently handle more I/O requests as the user uses more IONs. To exploit the I/O bandwidth of CloudBB and measure peak I/O throughputs, we increase IONs with matching number of CNs. Except in case of CNs is one and two, we use three IONs for 3-way replication.

Figure 3.21a shows the sequential I/O performance. As shown in the figure, CloudBB can accelerate the sequential I/O performance because throughput between CNs and IONs is much faster than the one between CNs and Amazon S3. At 64 CNs with 64 IONs, CloudBB achieves around 12 GiB/sec for read and 11.7 GiB/sec for write while `s3fs` only exhibits 4.3 GiB/sec for read, and 1.2 GiB/sec for write. Figure 3.21b shows aggregate I/O rates of random I/O, i.e., I/O operations/seconds (ops/sec). As shown in the figure, CloudBB exhibits higher I/O rates than Amazon S3 for the same reason as the sequential I/O. As we mentioned in Section 3.1.3, we use different threads in synchronization of replication and regular processes, we can see from the figure, the impact of synchronization is

total hidden in write. Although the I/O performance is high enough, CloudBB can provide higher I/O performance if the user disables file replication. In theory, as we use more IONs, CloudBB can provide more I/O throughput as well as I/O rate.

Metadata Operation

Metadata operations occur whenever initiating I/O operations. Achieving *fast* as well as *scalable* metadata operations is critical to increase overall I/O performance of data-intensive applications.

Figure 3.21c shows metadata operation rate (ops/sec). We choose file creation operations since it has been widely used to evaluate metadata performance. In the evaluation, we create 1,000 files in total under the different number of MNs and CNs. As shown in Figure 3.21c, CloudBB can significantly improve file creation performance. For example, the performance in CloudBB with 2 MNs is 518 times faster than Amazon S3 in 16 CNs. Meanwhile the metadata operation rate of Amazon S3 is significantly low, and the numbers are hardly visible in the figure. The metadata performance with a single MN hits its peak performance (i.e., 17,000 ops/sec) in 32 CNs since aggregate file creation rate from the CNs exceeds a rate that a single MN can handle. However, by increasing the number of MNs to 2, we see further improvement up to 27,000 ops/sec in 64 CNs, which is almost 2x faster than the one with a single MN. Similarly, with 4 MNs, metadata performance can achieve up to 35,000 ops/sec in 128 CNs. From these experiments, we show that CloudBB can significantly improve metadata performance by more than two order of magnitude compared to Amazon S3. Our hybrid model can alleviate the bottleneck caused by the Master-worker model in traditional storage systems.

As described in Section 3.1.1, metadata operations are distributed across MNs. As the user increases the number of SCBBs, CloudBB can concurrently handle more I/O requests. Thus, CloudBB is not only fast but also scalable.

From these comprehensive I/O evaluations, we see that CloudBB exhibits high I/O performance as well as scalability. Since I/O operations issued by data-intensive HPC applications are combinations of these I/O patterns, CloudBB can generally improve I/O performance of any applications by simply mounting our CloudBB filesystem.

Table 3.4: Input Data Size and Total IO size of Applications

Benchmarks	Input size (GB)	Total I/O size (GB)
Montage	1.2	28.1
Supernovae	2.4	5.0
BT-IO from NPB	No input data	6.8
Himeno Benchmark	No input data	1.3 * N
Miranda_IO	No input data	23.2 * N

3.4.2 Failure Recovery

CloudBB also supports fault tolerance capabilities such that CloudBB creates file replicas across different IONs and restore files on a failure. We evaluate how fast CloudBB recovers from an ION failure.

Figure 3.22a shows the write performance of a single CN in one second of a time period with a single SCBB consisting of four IONs and MNs. We use four IONs to ensure three replicas even after a failure. To inject a failure to CloudBB, we kill the CloudBB-related process on the primary ION that stores the file CN writing to, at 0.35 second time point. CloudBB initially stores zero to 1,000 of files whose size is 5 MB each, i.e., the total sizes are zero to 5 GB.

As shown in the figure, the write performance drops to zero MB/sec after the ION failure, and then returns to the same throughput after completing the recovery. The downtime is dominated by file recovery time. Thus, the downtime becomes longer as the failed node stores more files. However, even in 5 GB, CloudBB can still quickly recover from the failure within 0.25 seconds. It is because CN not need to wait for the recreation of replicas finished before restarting data transfer to the ION. As mentioned in Section 3.1.3, the MN promotes one of the secondary replica to primary replica after the previous one fails, as long as the procedure of replica promotion finishes, the write could be resumed. Since the replica promotions are performed only inside of the MN, the impact of ION failure here is extremely slight. The total size of files stored on a failed ION decreases as we use more IONs in CloudBB. Therefore, CloudBB can recover more quickly with more IONs.

Table 3.5: I/O size of Montage and Supernovae

	Montage	Supernovae
Input data set size	1.2G	2.4 GB
Total I/O amount	28.1G	5.0 GB
Read size:Write size	2.6	1.0
Total I/O amount to intermediate files	27.4 GB (97.7%)	4.2 GB (83%)

3.4.3 Case Studies in Real Applications

Performance

In the previous sections, we presented effectiveness of CloudBB in the comprehensive I/O patterns. In order to illustrate how such improvements impact on real applications, we evaluate CloudBB with five real data-intensive HPC applications and benchmarks as follows.

- Montage (Strong scaling) [14]: An astronomical image mosaic engine developed in NASA. Montage is a workflow application constructing custom science-grade mosaics by composing multiple astronomical images.

- Supernovae (Strong scaling) [19]: A astronomical workflow application used to find supernovae candidates from images taken by telescope.

- BT-IO (Strong scaling) [61]: BT-IO is a Block tridiagonal solver in NPB benchmarks [15]. Abbreviation of Block-Tridiagonal program provided by NPB benchmarks. We use *MPI-IO full* mode in which each process issues N-1 write to a single shared file via MPI-IO.

- Miranda_IO (Weak scaling) [58]: Miranda_IO is the I/O kernel from the Miranda hydrodynamics application code. Miranda_IO requires four processes at the minimum.

- Himeno benchmark with checkpointing (Weak scaling): We added a coordinated checkpointing routine to Himeno benchmark [9]. Himeno benchmark is a stencil benchmark solving Poisson's equation using the Jacobi iteration method. The

Extended Himeno benchmark runs for 100 iterations. We write a checkpoint every 10 iterations, which is about 4 seconds.

For strong scaling applications, we use the same data set with different number of CNs, i.e., the total problem size is fixed. For weak scaling applications, we fix problem size per CN. The sizes of the data set of each application are shown in Table 3.4. For executing the two workflow applications, Montage and Supernovae, we use GXP [7] for the process management.

The `s3fs` filesystem also supports metadata caching on CNs. However, this option does not guarantee that a file written by a process on a node is immediately visible to a process on another node even if first process closes the file. In addition to the *eventual consistency* problem described in Section 2.4.2, the inconsistent metadata cache (*metadata consistency* problem) also affects application executions, and applications occasionally failed in our preliminary evaluations. Thus, in the rest of experiments, we disable the metadata cache option for `s3fs` in order to ensure that the applications can correctly run. CloudBB synchronizes metadata cached on CNs with metadata on MNs whenever files are closed, CloudBB does not have the metadata consistency problem.

Figure 3.22b, 3.22c and 3.23a show execution times of Montage, Supernovae and BT-IO. Since the total I/O size is fixed in this evaluation, we also use the fixed number of MNs and IONs, one MN and three IONs respectively. BT-IO requires the number of process to be n^2, so we run BT-IO in different scales to Montage and Supernovae.

In workflow applications, each process exchanges intermediate results via files, i.e., a file written by one task is immediately read by the next tasks. This means workflow applications have high temporal I/O locality as shown in Table 3.5. Since these intermediate files are cached on IONs, CloudBB improves the I/O performance. Especially, I/O in Montage is much more intensive than Supernovae. Thus, execution times of Montage can be significantly reduced more than Supernovae.

In BT-IO, we only show the performance of CloudBB because N-1 write patterns do not work on Amazon S3 as described in Section 2.4.2. On the other hand, different portions of a single file can be updated by multiple processes in parallel in CloudBB with non-file caching mode.

Next, we show the results of the weak scaling applications. Since the total I/O size increases as the number of CNs grows in weak scaling, we also increase the number of IONs to accommodate the increasing I/O requests while keeping the ration of the number of IONs to the number of CNs 0.5. For example, when we run the weak scaling applications with 128 CNs, we use 64 IONs. However, we use three IONs at

the minimum for the 3-way replication. Thus, we use three IONs for executions with one, two and four CNs.

Figure 3.23b and 3.23c show execution time of the stencil benchmark and Miranda_IO. Since Miranda_IO requires four processes at the minimum, Figure 3.23b begins from four CNs. As shown in the figures, execution time with CloudBB is almost constant even with increasing number of CNs in both scenarios. CloudBB is scalable by appropriately increasing the number of IONs. In Miranda_IO with 128 CNs, especially, the execution time with CloudBB is 28.7 times smaller than the one without CloudBB.

Overall, we see that CloudBB can improve the I/O performance as well as exhibit scalability for data-intensive applications. In general, throughput of shared cloud storage (level-2 storage) is much lower that network bandwidth between cloud instances (level-1 storage). Therefore, CloudBB is expected to improve I/O performance on other cloud platforms. In that sense, our CloudBB can be also applied to any cluster systems including supercomputers.

Cost

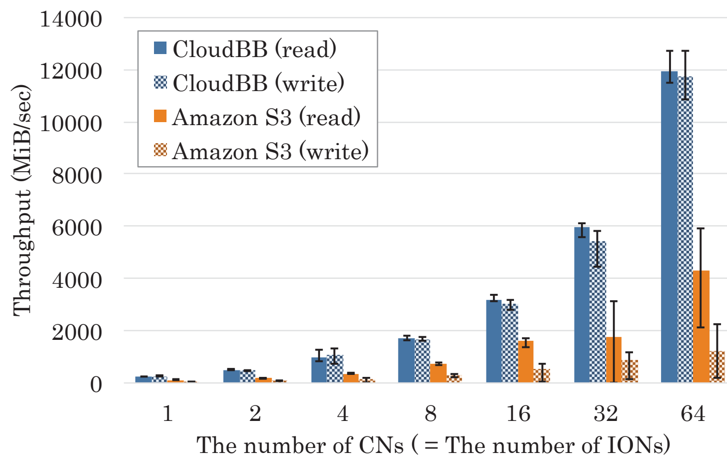
Although main focus of CloudBB is to improve I/O performance, we see that CloudBB also contributes to reduce expense for running cloud instances during the experiments. Figure 3.24a, 3.24b, 3.24c and 3.25 show the costs for executing Montage, Supernovae, Miranda_IO and the Himeno benchmark in Amazon EC2 based on the prices shown in Table 3.3. The cost includes expense for running MNs and IONs as well as CNs in CloudBB.

As shown in the figures, we can reduce the cost by using CloudBB in most of the cases, especially in large-scale executions. Especially, in Miranda_IO with 128 CNs and 64 IONs, we are able to not only improve I/O performance but also achieve 94.7% reduction of the cost. CloudBB can significantly reduce running time of instances. Therefore, CloudBB can also reduce the cost even with the additional instances of MNs and IONs.

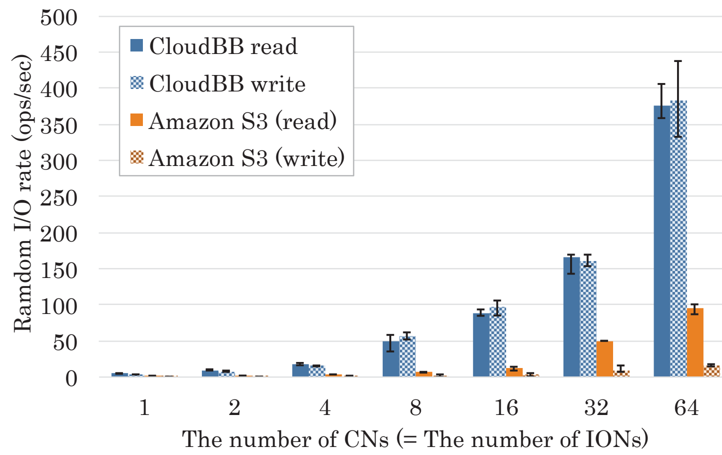
In general, there is a trade-off between I/O performance and the cost in CloudBB. If we can model the I/O performance of CloudBB and choose optimal number of IONs and MNs according to I/O workloads, we can achieve the good trade-off. We consider the modelling and the auto-tuning in future work.

Correctness

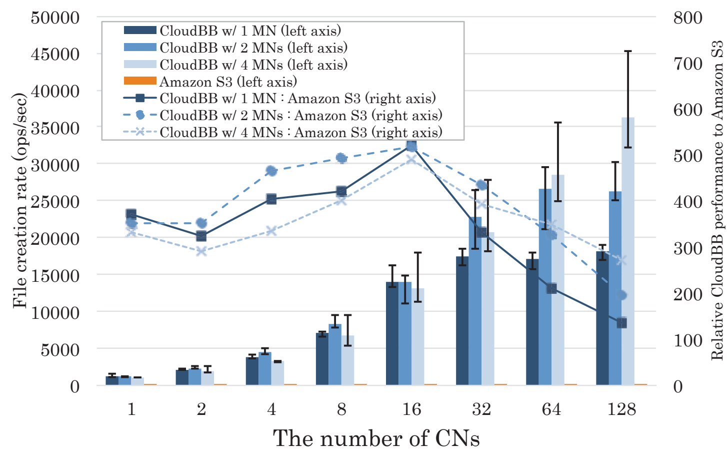
Another important thing to note is that Amazon S3 has consistency problems, *metadata consistency* and *eventual consistency*. We can avoid the *metadata consistency* problem by disabling the metadata cache option on s3fs, but *eventual consistency* problem still cannot be resolved as described in Section 2.4.2. Thus, all the applications cannot be guaranteed that the numerical results are correct when using only Amazon S3 [21] . However, by using CloudBB, we can guarantee that the final results are correct as described in Section 3.1.



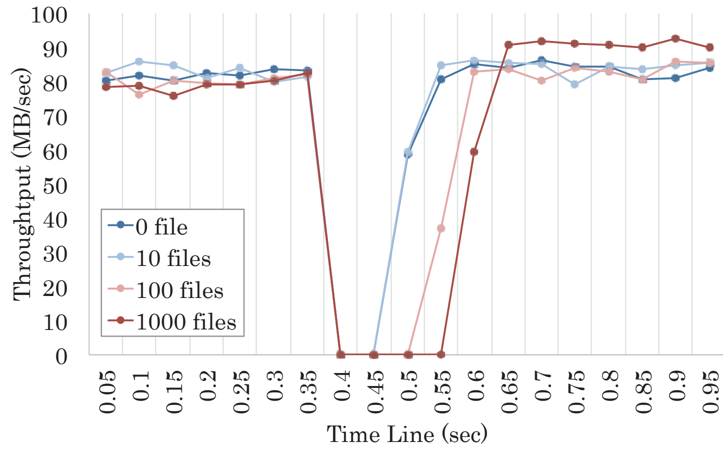
(a) Sequential I/O performance



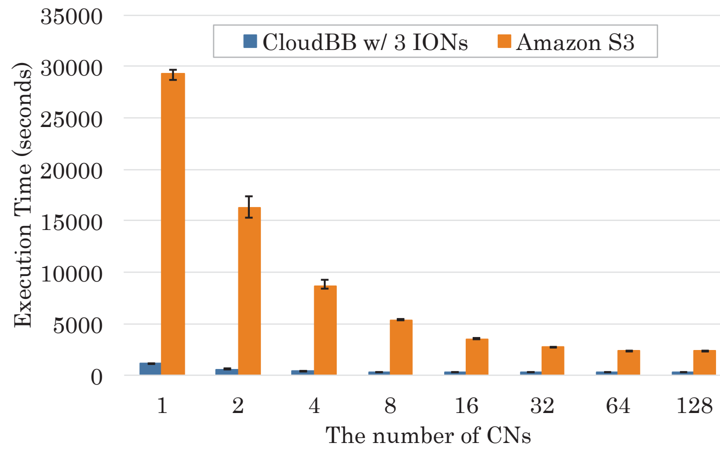
(b) Random I/O performance



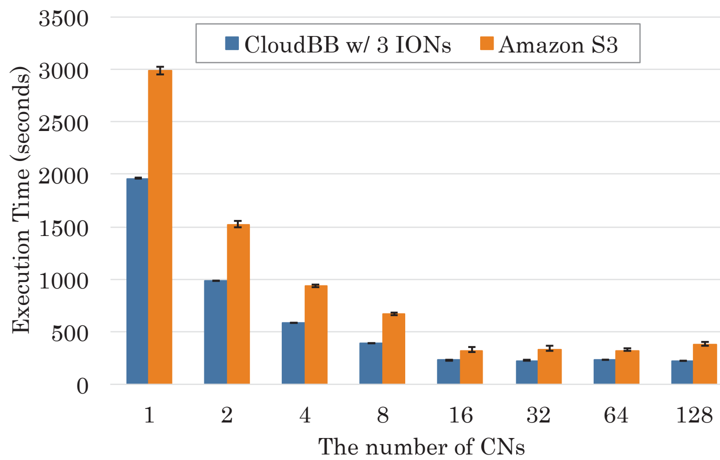
(c) File create performance



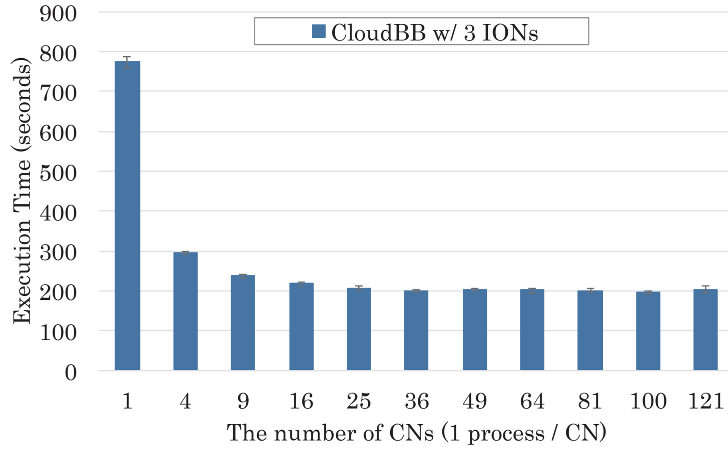
(a) Failure recovery



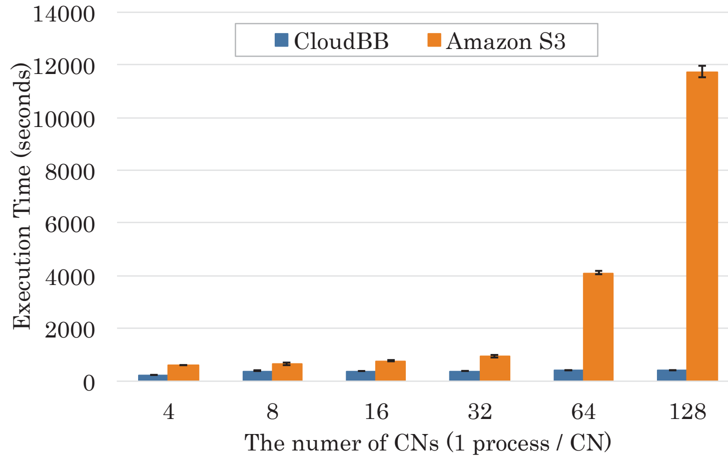
(b) Execution time of Montage using three IONs (Strong Scaling)



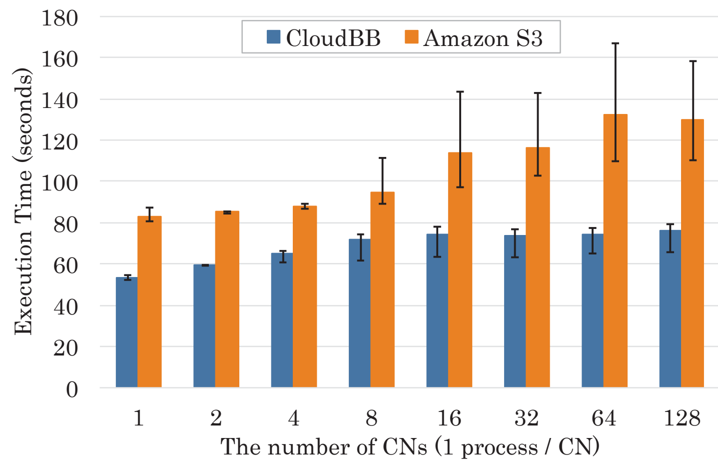
(c) Execution time of Supernovae using three IONs (Strong Scaling)



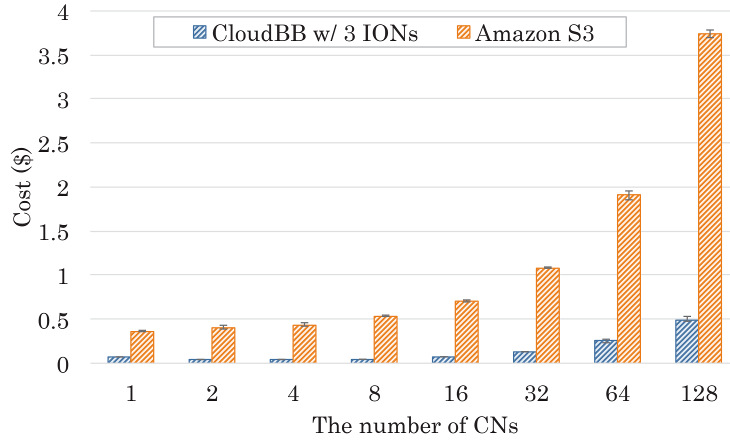
(a) Execution time of BT-IO using three IONs (Strong Scaling)



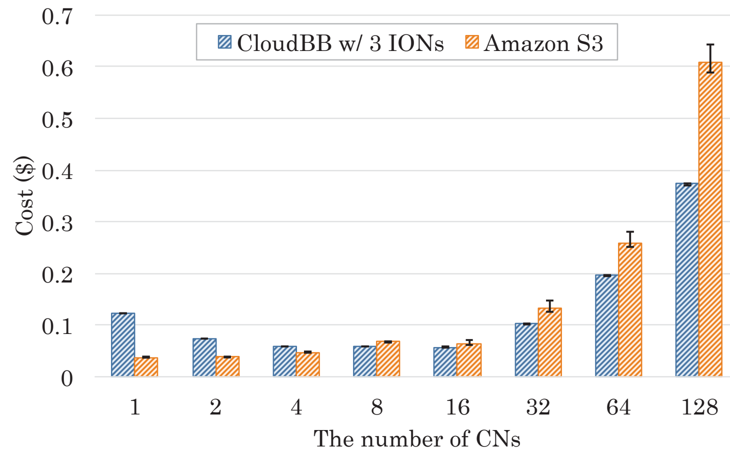
(b) Execution time of Miranda_IO (Weak Scaling)



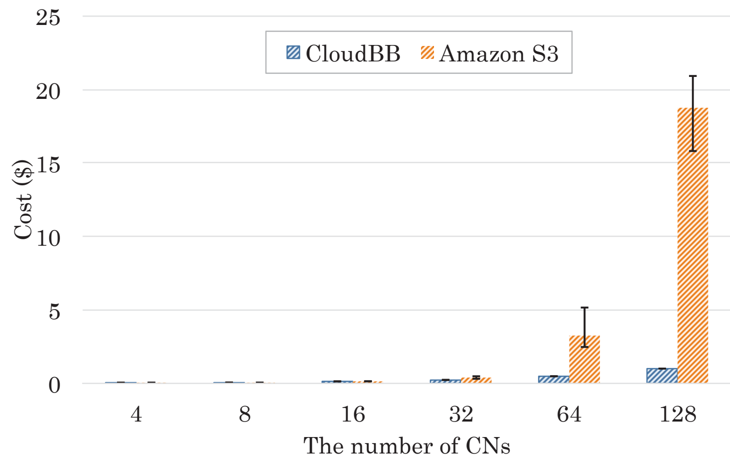
(c) Execution time of the Himeno benchmark w/ checkpointing (Weak Scaling)



(a) Cost for execution of Montage



(b) Cost for execution of Supernovae



(c) Cost for execution of Miranda_IO

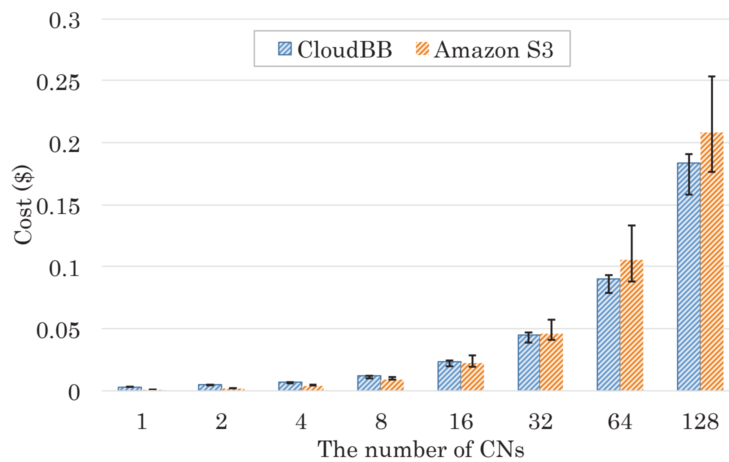


Figure 3.25: Cost for execution of the Himeno benchmark with Checkpointing

Chapter 4

Software-Level Burst Buffer System for HPC

As we mentioned, in order to help those HPC centers which do not high performance storage to easily improve the I/O performance and those with high performance storage to avoid resource contention, we extend our CloudBB work into HPC environments and propose HuronFS (Hierarchical, User-level and ON-demand FileSystem for HPC centers. In this section, we introduce our motivation in Section 4.1, and the major improvement from CloudBB in Section 4.2. Finally, we evaluate the HuronFS in Section 4.3.

4.1 Motivation of Having HuronFS

As we mentioned in Chapter 1, the computational performance of the HPC centers have dramatically increased, however, the performance of the parallel file systems can hardly catch up. Burst buffer systems have been proposed as a new tier between the compute nodes can parallel file systems to serve as a buffer layer and absorb the bursty I/O from the compute nodes to improve the I/O performance. However, due to the procurement and maintenance cost, only the latest cutting-edge HPC centers equipment the burst buffer systems. Moreover, for the existing systems, integrating the physical burst buffer systems involves additional procurement and deployments, hence it is not economically nor logistically feasible to add a burst buffer system to every HPC system. Moreover, physically deployed burst buffer has its designed capacity and performance, which can hardly adapt to any demand changes.

Hence we extend our CloudBB to the HPC centers to solve the problems. We propose a software- and user-level on-demand burst buffer system, HuronFS. Similar

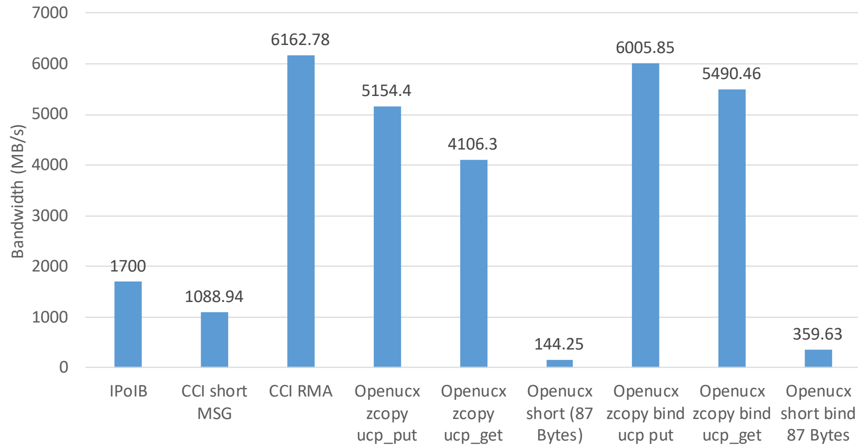


Figure 4.1: The Bandwidth Comparison between TCP/IP and RDMA

to the CloudBB, HuronFS leverages compute nodes to build a software-level burst buffer system, and as a user-level system, HuronFS can easily deployed on any HPC centr or can even be deplyed for a certain job by a user. Moreover, in order to support the large data buffering from the large-scale data-intensive applications, HuronFS supports limited buffer. By using the HuronFS, user can accelerate their applications even without a physical burst buffer system.

4.2 Improvement from CloudBB

As an extension of CloudBB into the HPC centers we made several improvement on the HuronFS compared to the CloudBB. In this section, we introduce all the improvements.

4.2.1 TCP/IP →CCI

The biggest improvement from the CloudBB is the communication protocol. In the CloudBB, in order to support all the cloud environment, we decided to use the widely supported TCP/IP protocol. However, when we port the CloudBB into the HPC environment, the communication protocol became the critical part that limits the performance. Unlike the Cloud environment, HPC centers equipment with high performance network such as Infiniband and Omni-path, using the TCP/IP protocol can hardly achieve the full performance of the network devices. In the Figure 4.1 and 4.2 we show comparison between the IPoIB, which is TCP/IP support over the

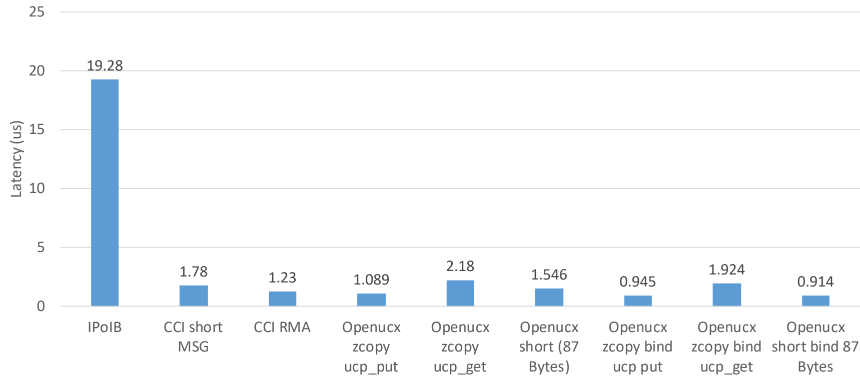


Figure 4.2: The Latency Comparison between TCP/IP and RDMA

Infiniband against the two high performance communications frameworks: CCI and Openucx, with native Verbs and RDMA supports. As we can see, the CCI and OpenuCX can achieve 3.6 times higher Bandwidth compared to the IPoIB and 21 times lower latency. Moreover, in addition to the Verbs from RDMA, CCI abstract the communication layer and supports the TCP/IP protocol and other network devices using the same code base. Therefore, CCI provides more portability. Hence, in order to fully utilize the high performance network, we ported the communication layer of CloudBB from the TCP/IP to CCI.

4.2.2 Multi-threading Support

In the CloudBB, all the communications and request handling are done by a single thread for simplicity. However, in the HPC world, it is well known technique that using multiple threading can overlap the communication and computation. By having the request handling and data transfer overlapped, we can further improve the performance of our system.

In order to generalize the multi-threading model for multiple different communication jobs as well as minimize the overhead of multi-threading, we use a lockless-consumer and producer queue to exchange the data between threads.

4.2.3 Limited Buffer Support

In the HPC fields, users need to handle GiBs to TiBs of data in a single job. In the design of CloudBB, we assume that all the data can be buffered in the main memory. However, such assumption cannot hold in the HPC centers. Hence we need to deal

Table 4.1: TSUBAME 2.5 Specifications

System	TSUBAME 2.5
CPU	Intel(R) Xeon(R) CPU X5670 @2.93GHz 12 cores 2 sockets
Memory	54 GiB
Network	Mellanox 4X QDR InfiniBand 40 Gb/sec
Storage	Lustre

Table 4.2: TSUBAME KFC Specifications

System	TSUBAME KFC
CPU	Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
Memory	62 GiB
Network	Mellanox FDR InfiniBand HCA 54.
Storage	NFS
Nodes used	38

with the situation that the required buffer is more than the main memory available in the IOnodes. In the HuronFS, similar to the cache system in the CPU, we introduce the data swapping into the HuronFS. When there is enough space in the burst buffer, then the data transfer is performed as normal. When there is not enough space, an old data is select to be swapped out, after freeing the space, the new data can be stored in the burst buffer. Asynchronous data write back is always preformed on the background.

4.3 Evaluation

We evaluate the performance of HuronFS on two HPC centers, TSUBAME 2.5 and TSUBAME KFC. The specifications of the two systems are shown in Table 4.2 and 4.1

Table 4.3: HuronFS Configurations

# of Master	1
# of IOnode	1
IOnode Capacity	32 GiB
Client Buffer	100MB

Table 4.4: Basic I/O Benchmark Details

Benchmark	File Size	Transfer Size
Sequential	10MB per process	10MB
Random	1KB per process	8 Byte

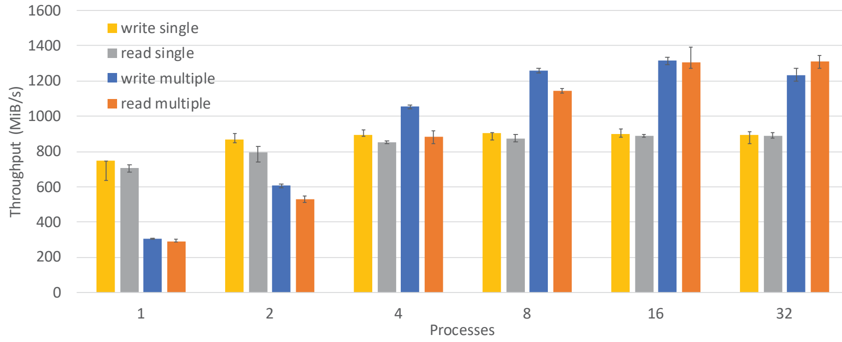


Figure 4.3: Sequential I/O Performance with Single Client on TSUBAME 2.5

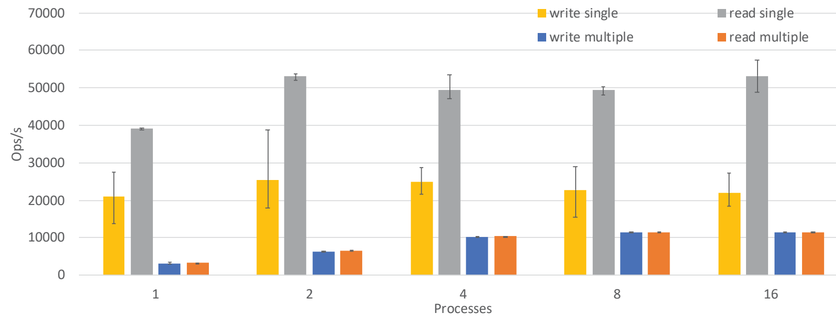


Figure 4.4: Random I/O Performance with Single Client on TSUBAME 2.5

The two HPC systems have different storage system and different network interface. We use the configuration in Table 4.3 for our HuronFS.

We run two types of evaluations on the two systems : basic I/O performance, including sequential, random I/O, and metadata performance; real applications. By evaluating and comparing the performance of the two evaluations against the performance of the original storage system, we demonstrate the effectiveness of our HuronFS. For the basic I/O performance, we use a IO benchmark written with MPI, IOR [72] provided by LLNL. We run both sequential and random I/O test with IOR benchmark with the configurations shown in Table 4.4. For the metadata performance, we again use a benchmark provided by LLNL, mdtest [56], to test the file creation performance as a representative of the metadata performance. We set the mdtest to run in a weak scaling manner and create 160 file per process.

4.3.1 Basic I/O Performance

First of all, we test the basic I/O performance of HuronFS on TSUBAME 2.5 supercomputer. We run the basic I/O test in two different ways, single node and multiple nodes. As the name indicates, in the single node run, we use a single client,

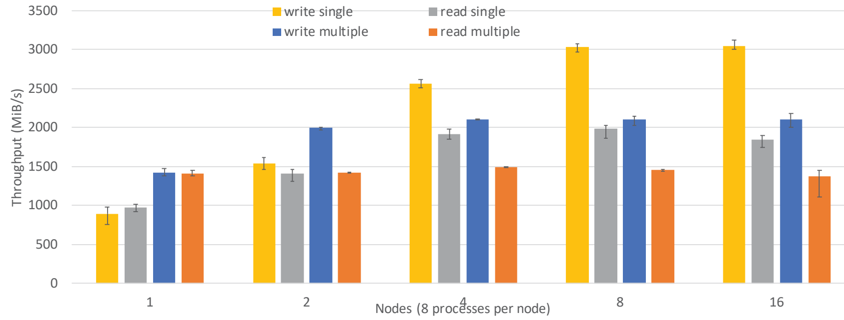


Figure 4.5: Sequential I/O Performance with Multiple Clients

and increase the number processes on the node to find out the saturation point. This experiment help us to find out the maximum performance we can achieve with a single client. Then, we fix node process number to the minimum number of processes that saturate single node performance, and increase the number of clients to show the maximum performance of a single server. For each of the experiments, we show two sets of the evaluations: single thread vs multiple thread mode of HuronFS. We show the performance difference between the two modes and the performance trend when scaling up.

Figure 4.3 shows the sequential I/O performance of a single node. As we expected, for the single thread mode, since the FUSE uses only one thread to handle the I/O request, the overall performance does not improve much even with more processes. On the other hand, with multiple threads in FUSE, when the number of processes in small, e.g. 1 process, the performance is lower than single thread mode due to threading overhead. However, the performance increases as the number of processes increases, exceeds the performance of single thread mode and plateaus at 8 nodes per node.

Figure 4.4 shows the random I/O performance. We observe a difference performance trend, as the single thread performance surpasses the multiple thread mode in all the cases. That is because of the lack of buffer supports in multiple thread mode and the overhead of multiple threading.

Secondly, we scale our experiment up to show the maximum performance we can achieve with single server. We use the saturate point of 8 processes per node. Figure 4.5 shows the multiple clients performance on TSUBAME 2.5. As we can see, although the multiple thread mode gives us more performance in the single node case, however, due to the frequent small I/Os, the maximum performance we can achieve with a single server is lower than the single thread mode with much larger

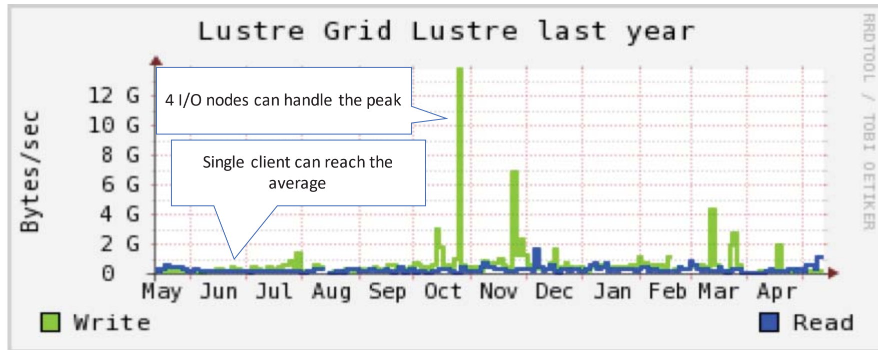


Figure 4.6: Luster Workload on TSUBAME 2.5

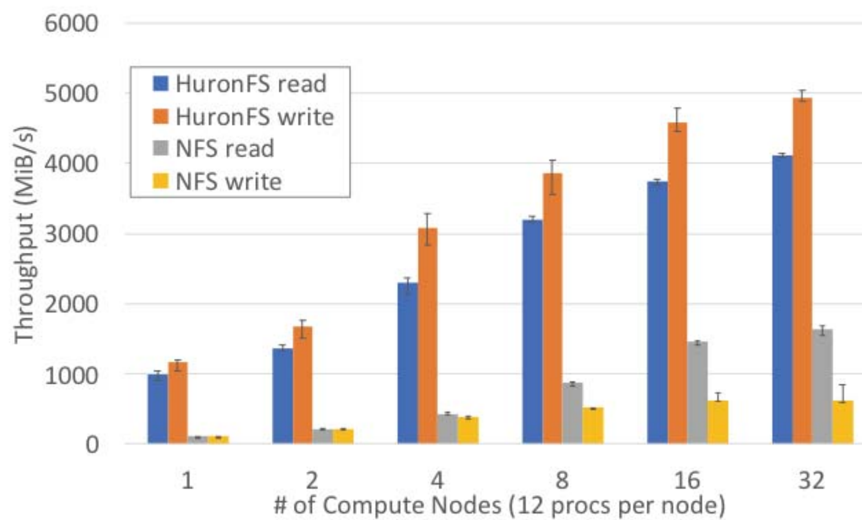


Figure 4.7: Sequential I/O Performance with Multiple Clients on TSUBAME KFC

I/O transfer size. With the single thread mode, we can finally achieve 3GiB/s from a single server.

In order to see if the performance of HuronFS is fast enough for large-scale data-intensive applications, we show the monitor results of the Lustre system on TSUBAME 2.5. As we can see from Figure 4.6, a single client can easily reach the average throughput requirement from the entire system while a single server can handle even the peak requirement from the system. Hence, from the experiments we conduct, we can see that HuronFS can achieve comparable performance to the parallel file system in the HPC centers, and help the users to achieve high performance when the system is under high resource contention.

We also show the same experiments on the TSUBAME KFC in Figure 4.7 4.8. In this experiment, we add the comparisons against the NFS on TSBUAME KFC. From the Figure, we see similar performance results to the TSUBAME 2.5, and thanks to

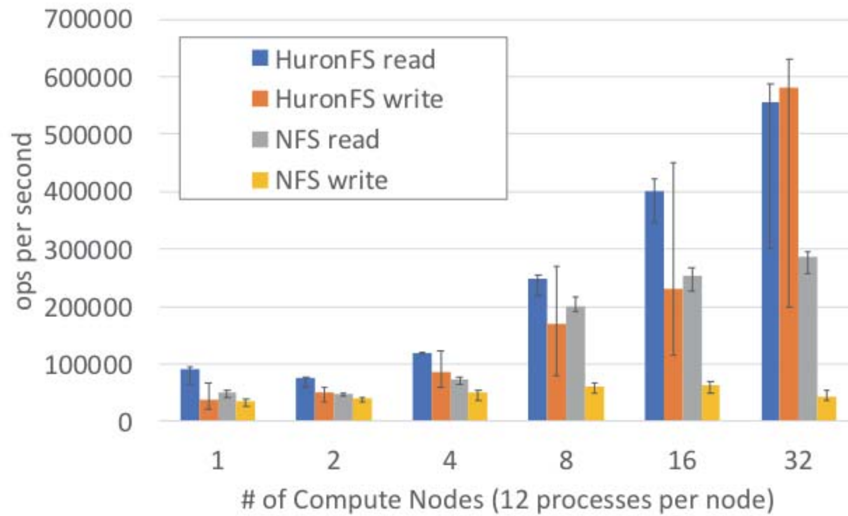


Figure 4.8: Random I/O Performance with Multiple Clients on TSUBAME KFC

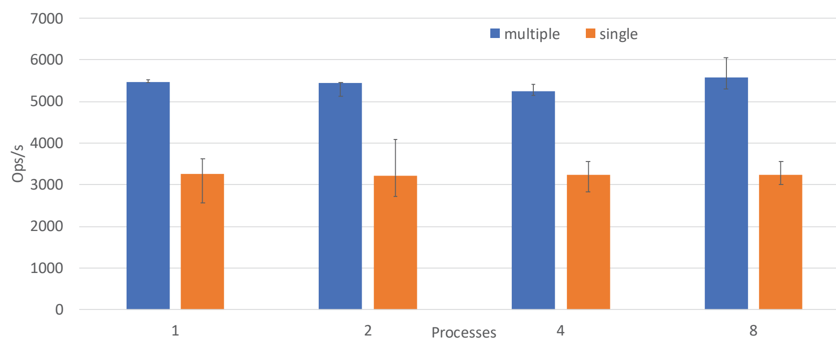


Figure 4.9: File Creation Performance with Single Clients on TSUBAME 2.5

the higher performance network, we can achieve around 5 GiB/s from a single IO server. We also achieve nearly 7.8 times performance improvement compare to the NFS on TSUBAME KFC. From these experiments, we can say that for those HPC centers that do not have fast parallel file systems, using HuronFS can greatly improve the I/O performance from the system.

4.3.2 metadata performance

Besides the sequential and random I/O performance, we also evaluate the meta data performance of HuronFS. Similar to the sequential I/O case, we first measure the meta data performance with single client and then scale up.

Figure 4.9 shows the single client results, as we can see, the multiple thread mode of FUSE helps to accelerate the metadata performance. However, scaling up from a

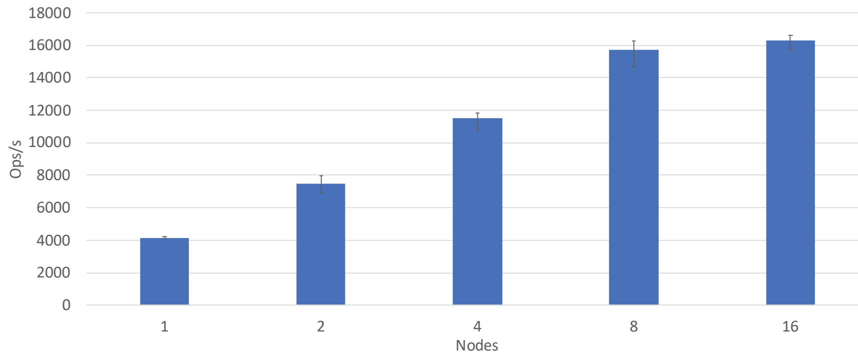


Figure 4.10: File Creation Performance with Multiple Clients on TSUBAME 2.5

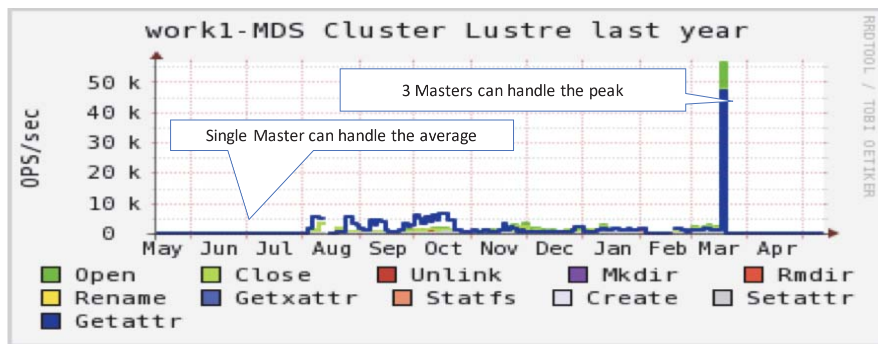


Figure 4.11: Lustre Metadata Workload on TSUBAME 2.5

single node does not help to improve the meta data performance. With a single node, we can achieve around 5500 ops.

Figure 4.10 shows the multiple client results of meta data. Similarly, by scaling up, we show the maximum metadata performance of a single master. As we can see, by increasing the number of clients, we can finally achieve around 16000 ops from a single master, which is 3 times more than with a single client.

We compare the average workload on the TSUBAME 2.5 LUSTRE to see if HuronFS can achieve a good enough performance. From the Figure 4.11, we see that with a single master, we can handle the average metadata requests even from the entire system. With 4 masters, the peak requirements can be handled. From these experiments, we can say that in addition to the basic I/O performance, HuronFS can also provide comparable metadata performance.

4.3.3 Real Applications

In the previous section, we demonstrate that with HuronFS, we are able to improve the basic I/O performance on supercomputer. In this section, we evaluate how

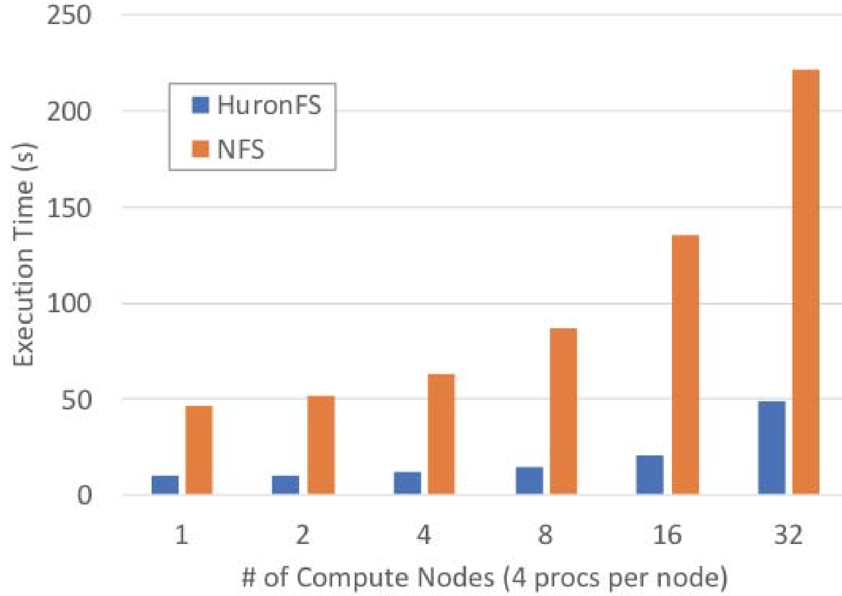


Figure 4.12: Miranda I/O results on TSUBAME KFC

Table 4.5: Real Application Details

Applications	Output File Size (MB)	Processes
Miranda IO	110 per node	4 per node (weak scaling)
BTIO	6400	64 processes (strong scaling)

HuronFS actually improves real application performance with the two extension to CloudBB: High performance network support and limited buffer support. We first evaluate the real application performance against NFS in section 4.3.3, then we evaluate how the application performance changes with different size of buffers in section 4.3.3.

HuronFS to improve the performance

In addition to the basic I/O performance and metadata performance, we also evaluate the performance from the real application and compare them against the NFS on TSUBAME KFC to show how the performance improvement we get from the basic I/O performance can accelerate real applications. We evaluate with 2 real applications : Miranda I/O from LLNL and BTIO from NAS Parallel Benchmark with the specifications in Table 4.5. Figure 4.13 and 4.12 show the performance comparison of Miranda I/O and BTIO respectively. As we can see, with the basic I/O performance improvements, we can actually improve the run time of real applications. Moreover, the higher improvements in the weak scaling experiment

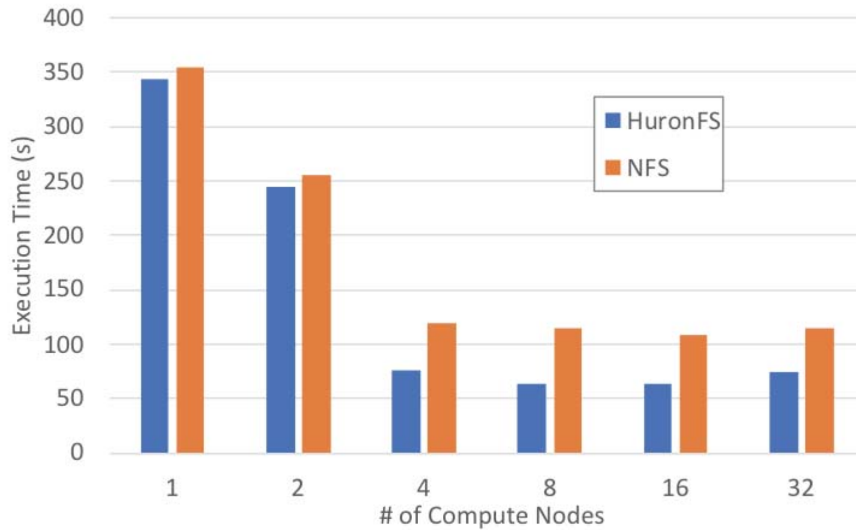


Figure 4.13: BTIO results on TSUBAME KFC

Table 4.6: Real Application Details

Applications	Total Access Size (MB)	Processes
Miranda IO	444*N	4 per node (weak scaling)
BTIO	6,800	64 processes (strong scaling)
Montage	7,500	12 per node (strong scaling)

with Miranda I/O, we can see that with more I/O, the more improvement we can achieve with the higher performance from HuronFS.

From the experiments we conducts with basic I/O, metadata, and real applications on two supercomputer systems, we show that by using the HuronFS, we can achieve comparable performance to the state-of-art parallel file system in the HPC centers, and help users to still enjoy the high I/O in the HPC centers even in the case of I/O contention. Moreover, for those do not have a fast parallel file systems, using HuronFS can easily improve the I/O performance of the system. By evaluating with the real applications, we demonstrate that the improvements from the I/O can actually help the real applications. The more I/O workload the applications issues, the more improvement they can achieve by using the HuronFS. Hence, in addition to the Cloud environment, our software-level burst buffer system can also help in the HPC centers.

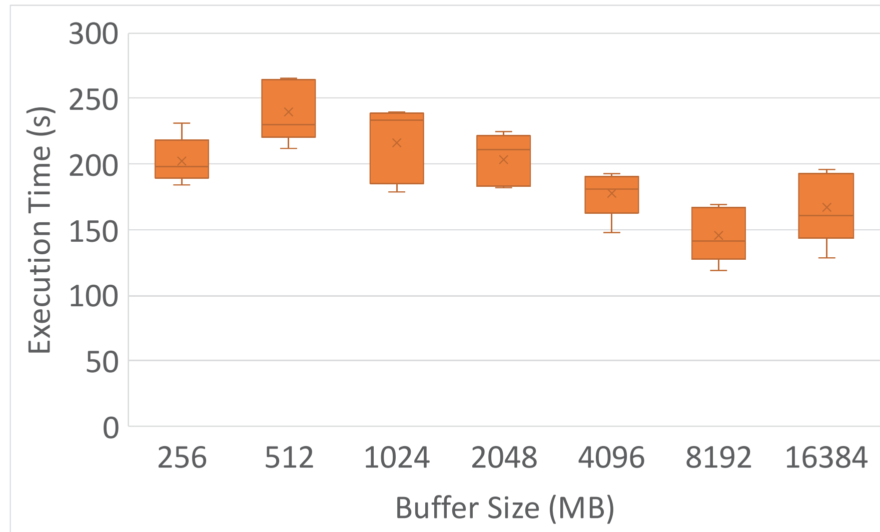


Figure 4.14: Miranda I/O with Limited Buffer on TSUBAME KFC

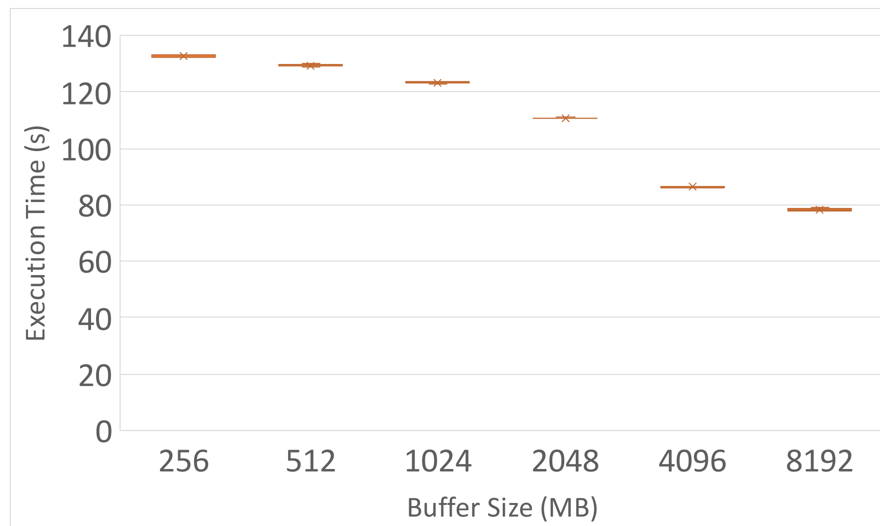


Figure 4.15: BTIO with Limited Buffer on TSUBAME KFC

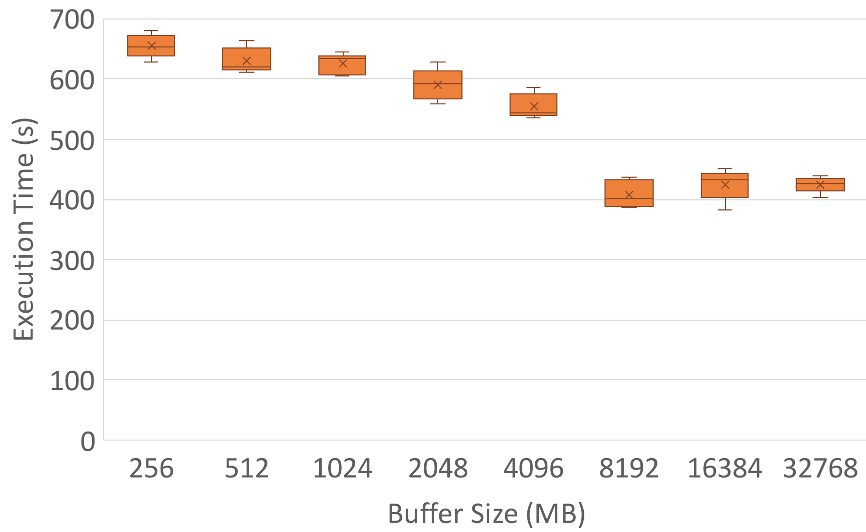


Figure 4.16: Montage with Limited Buffer on TSUBAME KFC

Applications with Limited Buffer

In this section, we show how applications performance changes with limited buffer. We evaluate with three applications, montage, miranda IO and btio with the parameters in Table 4.6.

As we can see from Figure 4.14, 4.15, 4.16, HuronFS supports the applications with limited capacities far less than the application access size. With more buffer, HuronFS can also achieve higher performance, however, with limited buffer, the performance reductions are not significant.

From these experiments, we can see that using HuronFS, we can accelerate basic sequential and random I/O as well as real applications. Moreover, HuronFS can supports applications with far less capacities with limited performance penalty.

Chapter 5

Analysis of the Configuration of Burst Buffer

5.1 Design In Simulation

In this section, we discuss the design idea and choices included in our simulation. We choose to build our simulator based on the trace-driven simulation approach for usability and generality. (Section 5.1.1) As the critical part of the trace-driven simulation, trace, we choose a FUSE-based tracer, MUSE, for multiple advantages. (Section 5.1.2

5.1.1 Trace-driven Simulation

To address these research questions, we simulate the performance of applications under different burst buffer configurations. Many researches have been done on simulations with different simulation techniques. There are three common approaches of simulation techniques: trace-driven simulation, execution-driven simulations, and modeling. Trace-driven simulation requires a real system to run the applications, obtains the trace of the target operations, and simulate based on the trace. On the other hand, execution-driven simulation emulate the real system, with the capability to execute the application directly and record all the events and performance data. Modeling builds performance models for one or a group of applications for calculating the overall performance with a set of parameters. We carefully compare the advantages and disadvantages of all the techniques, and we decided to build our simulator upon the trace-driven simulation techniques for the following benefits.

- **Usability**: Modeling technique requires the application developers to analyze their applications, and provides performance models. However, it is difficult to accurately model all the operations inside applications, especially when the applications are huge and complicated [48, 60]. Moreover, modeling requires verification and validation to ensure the correctness of the models, which further increase the difficulty. On the other hand, trace-driven simulation requires no comprehensive techniques nor complicated operations to apply to any applications [24, 27]. Instead, only the traces are required by executing them on the real systems.
- **Application Independence**: To help both normal users and system administrators to determine the right configurations of the burst buffer to use for their applications. Hence, we need an application independent solution for different kinds of applications. Modeling creates a unique performance model for each application, therefore different applications require different performance models to match, which makes modeling highly application dependent. On the other hand, tracing views applications as black boxes, and only deal with a set of given events from the applications. So tracing is more application independent and matches our design.
- **Generality**: Besides the application independence, our solution should also be platform independent, i.e. generality. We want to easily apply our solution on different clusters and different machines instead of a certain given group. Execution-driven simulator emulate the whole execution of the application on the target system. Hence, it requires a deep knowledge of the target system to design a simulator that matches the target system as detailed as possible [52–54]. Such characteristic makes execution-driven simulation highly platform dependent, i.e. a new study and simulator are required for a new system. Meanwhile, trace-driven simulation relies on the actual runs on the target system to include the features of the platforms into the simulation. By tracing and benchmarking the target system, the performance of application can be easily simulated on a totally new system [27].

5.1.2 FUSE-based I/O Tracing

As we mentioned in the previous section, we choose to build our simulator using trace-driven model. Tracing is the beginning and the essential part of a trace-driven

simulator. Many researches have been done on the I/O tracing of the applications with many existing I/O tracing tools [29,47,50,82,83]. In this research, we choose to use a FUSE-based I/O tracing tool, MUSE [75], to obtain the I/O trace for following four reasons:

Tracing all kinds of applications

As we mentioned, we want to provide a solution to trace all kinds of applications with all kinds of the I/O. Many I/O tracing tools [29, 82] have been designed to trace MPI, a widely used standardized and portable distributed programming model used in high performance computing for communication and I/O. In MPI model, all the processes call `MPI_INIT` in the beginning and end with `MPI_FINALISZE`. MPI I/O tracing tools match such programming model, initialize the tracing in `MPI_INIT`, produce the summary and generate the trace file during `MPI_FINALISZE`. However, such tracing model cannot be applied to all the applications, for example, workflow applications, where processes start and end at different time. On the other hand, FUSE is implemented as a kernel module, and independent from applications. FUSE traces all the I/O under a certain directory, i.e. the mount point, by letting all the I/O operations issue towards the mount point, FUSE based I/O tracer can trace all kinds of applications.

Tracing all kinds of I/O

Many I/O tracing tools leverage `LD_PRELOAD` or recompiling/relinking to intercept I/O calls from applications. However, I/O libraries usually have deep call hierarchy, while such techniques can only trace the first level function calls [47, 59]. For example, stream I/O library API `fclose` calls POSIX API `write` to flush the buffered data to the storage and `close` in its body. Moreover, applications can perform I/O using `mmap`, e.g. mmap a file into a memory region then update the memory area, file data would be updated to the storage by kernel [78]. Such I/O calls in other I/O libraries can not be simply intercepted by using `LD_PRELOAD` or recompiling/relinking. Furthermore, there are some system applications that cannot easily be recompiled, e.g. cat, tar, which would further limit the usage of such I/O tracing tools. Meanwhile, as we mentioned, FUSE is implemented in kernel, FUSE handles all the actual I/O data transfer request from the kernel. All buffering and I/O forwarding from high-level libraries are transparent to FUSE. In addition, as a kernel module, no recompiling/relinking is required to use FUSE, therefore, FUSE

Algorithm 1 Read Operations on our burst buffer model

1: **if** Data is not buffered **then**
2:     **if** Burst Buffer is full **then**
3:         Swap out an old data block according to the data replacement algorithm
4:     **end if**
5:     Read the data from the PFS
6: **end if**
7: Transfer the requested data to the computer nodes

Algorithm 2 Write Operations on our burst buffer model

1: **if** Burst Buffer is full **then**
2:     Swap out an old data block according to the data replacement algorithm
3: **end if**
4: Receive data from the computer nodes

based tracing tool is more general and able to trace all kinds of I/O from all kinds of applications.

Replayable Tracing

To perform trace-driven simulation, we require the trace file to be replayable.

- Each operation need to have its timestamp recorded to represent the order of the operations.
- Detailed information on each operation needs to be recorded, e.g. file name, access offset.

However, some tracing tools [50, 82] provide only summary of the I/O calls, e.g. number of read operations, which makes them unsuitable for our propose.

Tracing by everyone

In order to enable normal users to obtain the trace of their applications, we need to build our approach on user space. Some parallel file systems provide file system level I/O trace, with detailed information [55]. However, such information can not be accessed by normal system users easily. On the other hand, as a user space file system, FUSE provides non-privilege access to normal users.

5.2 Burst Buffer Model and Simulator

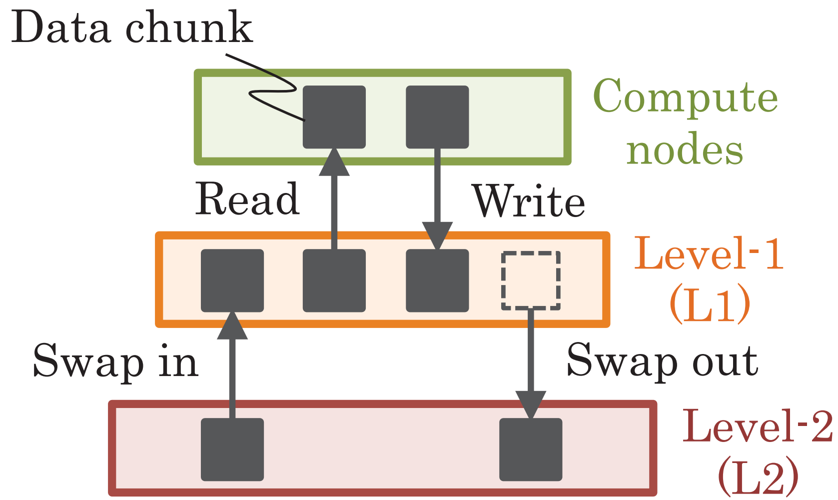


Figure 5.1: Burst buffer model

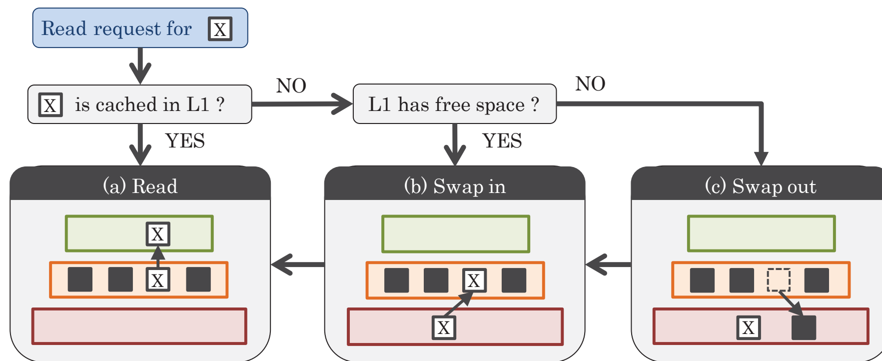


Figure 5.2: Read in the burst buffer model

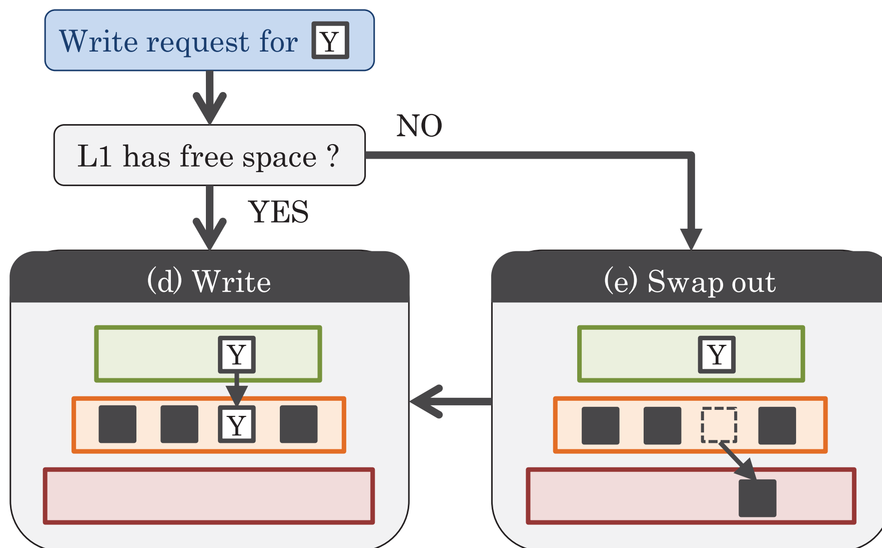


Figure 5.3: Write in the burst buffer model

Our burst buffer model consists of two-level storage, *level-1* (e.g., burst buffer) and *level-2* (e.g., PFS) as shown in Figure 5.1. Level-2 storage functions as persistent storage, while level-1 storage is cache space for level-2 storage and caches files of an application running on the *compute nodes*. Therefore, all I/O requests (i.e., read/write) are handled via the level-1 storage when compute nodes attempt to access files on the level-2 storage. In the model, level-1 has a finite capacity while level-2 has infinite capacity.

In our burst buffer model, files are divided into fixed-size *data chunks*. In reading data chunk X, the model first checks if chunk X is cached in level-1 (L1) storage. If cached, the chunk X is simply read from L1 (Figure 5.2-(a)). If not, the chunk X is swapped in L1 from level-2 (L2) storage (Figure 5.2-(b)). Then, compute nodes read the chunk X from L1. If there is no free space in L1 when swapping in, the model swaps out a chunk based on its data replacement algorithm such as LRU (Figure 5.2-(c)).

In writing data chunk Y, if there is free space in L1, the compute node will write chunk Y directly to L1 (Figure 5.3-(d)). If there is no free space, the model swaps out a chunk before writing to L1 (Figure 5.3-(e)). Our model also consider asynchronous write-back. *Dirty* data chunks, which are updated data chunks in L1 but not in L2, are written back to L2 storage while simultaneously handling other I/O requests. Whenever a chunk is updated, the model write the dirty chunk back to L2 storage in order to ensure consistency between L1 and L2 storage.

Our B2Sim simulates this burst buffer model.

We details our burst buffer model that B2Simsimulates. As we explained in the previous sections, we do not target on a particular burst buffer system, instead, we want to build our simulator as general as possible. To achieve this goal, we avoid any specific system designs and adapt a general burst buffer system model. Figure 5.1 shows an overview of our target burst buffer model. As shown in the figure, the model is fairly basic and general:

- We define burst buffer as a storage layer between the computer nodes and parallel file system
- Burst buffer is a group of nodes that provides data buffer with a higher bandwidth network
- The computer nodes run users applications and the burst buffer provides temporal data buffer while the parallel file system provides permanent data store

- All the I/O requests issued by computer nodes are forwarded to burst buffer, and the burst buffer transfer data with the parallel file system

However, in order to support applications with limited buffer size, we need to introduce a few more features, which have not been included into typical existing burst buffer systems, into our burst buffer model:

- All files split into fixed size data chunks
- Burst buffer has a given limited capacity, a swap queue is maintained by given data replacement algorithm
- When there is enough space remains, burst buffer acts as a typical burst buffers
- When the space runs out, burst buffer swap some data out according to the given data replacement algorithm to make room for the new data
- Data chunks that are consistent with these on PFS are marked as ”clean”, while those have been updated and are inconsistent with the PFS are marked as ”dirty”
- ”Dirty” data chunk needs to be updated to PFS before swapping out
- Burst buffer always updates ”dirty” data chunks to the parallel file system to reduce the overhead on data swap out

Similar to other cache systems, Algorithm 1 and 2 illustrate the read and write operations on our burst buffer model respectively . kentoAlso, explain Sync/Async mode

5.3 Simulation Methodology

In this section, we describe how our simulator is built on the top of the burst buffer model as a trace-driven simulator.

We build our simulator based on the burst buffer model described in section 5.2. At the end of the simulation, the simulator generates the estimated performance based on several performance counters, as detailed in section 5.3.1. However, in order to match the performance of a given application on a given system, the specifications of the application and system must be integrated into the simulator. Figure 5.4 shows the overview of our simulation. The key components of the simulation is the burst

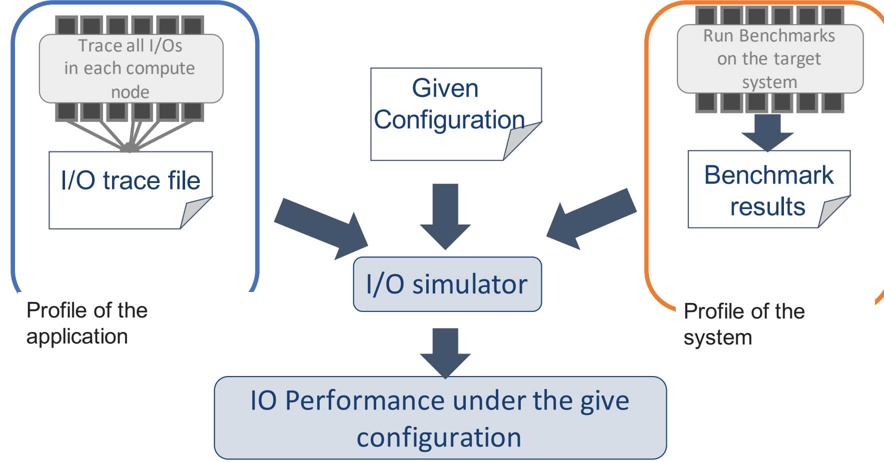


Figure 5.4: The overview of our simulation

buffer simulator. The simulator takes three inputs: the I/O trace of the applications (Section 5.3.4), which provides the I/O patterns of the applications; the benchmark of the system (Section 5.3.3), which provides the specifications of the target system; and the given configuration of the burst buffer.

A log parser in the simulator reads the trace files, and forwards all the read and write events to the burst buffer simulator one by one. Our simulator handles the events from the log parser and simulate the behavior of a burst buffer with the system performance specified by the system benchmarks. As the output, the simulator generates the expected I/O performance of the application under the given configuration of the burst buffer on the target system.

5.3.1 Performance Counters

$$Max_thr = \frac{Total_IO_Size}{\frac{Input_Size}{thr_{PFS}} + \frac{Total_IO_Size}{thr_{buffer}}} \tag{5.1}$$

$$Actual_thr = \frac{Total_IO_Size}{\frac{PFS_IO_Size}{thr_{PFS}} + \frac{Total_IO_Size}{thr_{buffer}}} \tag{5.2}$$

$$Total_IO_Size = IS + RHS + WS \tag{5.3}$$

$$Input_Size = IS \tag{5.4}$$

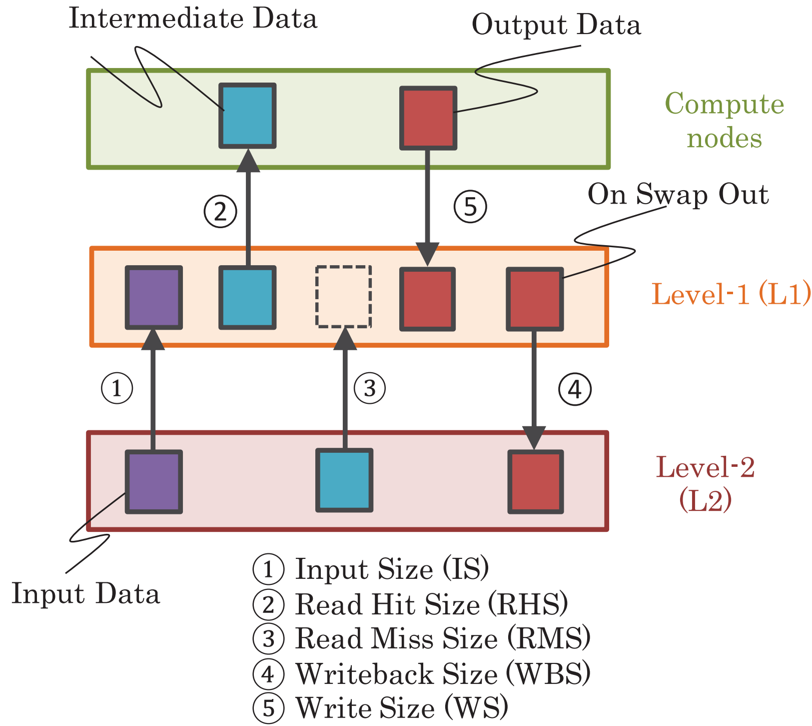


Figure 5.5: Examples of Performance Counters

$$PFS_IO_Size = RMS + IS + WBS \tag{5.5}$$

At the end of the simulation, we calculate the theoretical Max throughput using the Formula (5.1), which is the performance when using a burst buffer of unlimited size, and Formula (5.2) to calculate the actual performance under the given buffer size.

The RMS, RHS, WS, WBS, and IS are five different counters the simulator records:

- Input Size (IS) : records the total size of data that needs to be read from the PFS as input data. In the simulator, we assume no staging phase before the execution, hence IS records the total size of the input data.

- Read Hit Size (RHS) : records the size of data that is buffered in the burst buffer when it is read, i.e., the total amount of data that did not need to be swapped-in when being read.

- Read Miss Size (RMS) : records the size of data that have already been swapped out and need to be swapped in when being read. RMS is different from IS in

Algorithm 3 Read Operations In Synchronous Writeback

```
1: Input: read events from log parser
2: if Data is in buffer data pool then
3:    RHC+ = 1 {Read hit}
4: else
5:    RMS+ = 1 {Read miss}
6:    if Buffer data pool is full then
7:       Getting the last chunk from swap out queue
8:       if Chunk isn't clean then
9:          WBS+ = 1 {Writing data back to PFS}
10:      end if
11:   end if
12: end if
```

Algorithm 4 Write Operations In Synchronous Writeback

```
1: Input: write events from log parser
2: WS+ = 1 {Writing data to the burst buffer}
3: if Data isn't in buffer data pool then
4:    if Buffer data pool is full then
5:       Getting the last chunk from swap out queue
6:       if Chunk isn't clean then
7:          WBS+ = 1 {Writing data back to PFS}
8:       end if
9:    end if
10: end if
```

that RMS records the chunks that were once buffered in the burst buffer, but have been swapped out due to limits on the buffer capacities while IS doesn't.

- Writeback Size (WBS) : records the size of data chunks that has been updated and need to be written back to the remote PFS to make space for new read or write Operations.
- Write Size (WS) : records the size of data that has been written to the buffer. When the burst buffer runs out of the space, we also increase the WBS to compensate.

5.3.2 Two Writeback Strategies

To investigate their impact on performance, we simulate two common writeback strategies: synchronous and asynchronous writeback. In addition, to count the

Algorithm 5 Read Operations In Asychronous Writeback

1: **Input:** read events from log parser
2: $time = timestamp_now - timestamp_prev$
3: $write_back_size = time * Thr_PFS$
4: **repeat**
5: Popping writeback queue {writing data back to PFS in background}
6: **until** $write_back_size = 0$
7: **if** Data is in buffer data pool **then**
8: $RHC+ = 1$ {Read hit}
9: **else**
10: $RMS+ = 1$ {Read miss}
11: **if** Buffer data pool is full **then**
12: Getting the last chunk from swap out queue
13: **if** Chunk isn't clean **then**
14: $WBS+ = 1$ {Writing data back to PFS}
15: **end if**
16: **end if**
17: **end if**

writeback data accurately, for each data chunk, we assign a dirty flag, which is similar to the page writeback algorithm in the Linux kernel.

Recall that we have a writeback queue in our simulator to store the list of chunks that need to be buffered. However, when using synchronous writeback, dirty data chunks will not be inserted into the writeback queue, instead, chunk writeback only happens on swapping out of that chunk. Algorithm 3 and 4 shows the read and write operations under synchronous writeback strategy.

On the other hand, when using asynchronous write, writebacks are always performed in the background. Algorithm 5 and 6 shows the read and write operations under asynchronous writeback strategy. Dirty data chunks are inserted into writeback queue right after the updates. Each time a new event comes from the log parser, before handling the event, we calculate the amount of data that can be written back from the time interval between the previous operation and the throughput of PFS. Then we pop the exact amount of data chunks from writeback queue (line: 2˜6).

5.3.3 System Benchmark

A real system consists of many complicated components with complex interconnections. Each component and mode of interaction among components can have huge impacts on the performance and, as such, attempting to simulate or

Algorithm 6 Write Operations In Asynchronous Writeback

1: **Input:** write event from log parser.
2: $time = timestamp_now - timestamp_prev$
3: $write_back_size = time * Thr_PFS$
4: **repeat**
5: Popping writeback queue {writing data back to PFS in background}
6: **until** $write_back_size = 0$
7: $WS+ = 1$ {Writing data to the burst buffer}
8: **if** Data isn't in buffer data pool **then**
9: **if** Buffer data pool is full **then**
10: Getting the last chunk from swap out queue
11: **if** Chunk isn't clean **then**
12: $WBS+ = 1$ {Writing data back to PFS}
13: **end if**
14: **end if**
15: **end if**
16: **if** The data chunk is in writeback queue **then**
17: Putting chunk to the head
18: **else**
19: Inserting chunk to the head
20: **end if**

model such system without any abstractions is difficult and also limits the generality of the simulation/model. We hide such complexity while getting an accurate and generalizable view of the system by focusing on and benchmarking the I/O performance of the system.

We build an I/O benchmark to measure the I/O throughput. As shown in [44,73], the performance of the storage systems has can be affected by many configurations of the systems, such the stripping size and block size, which causes the performance of the storage to vary based on size of the data being transferred. Hence, instead of using a typical throughput + latency performance model, we benchmark the performance for different transfer sizes, ranging from 1KiB to few MiBs or GiBs, and use the closest measured results for any given transfer size in the simulation. Since we need to benchmark the throughput of some small transfer sizes, some metadata operations would also significantly affects the benchmark results, such as metadata operations for file `open`. Benchmarks like IOR [72] do not distinguish between the metadata operation into and data transfer performance in their results. To avoid these metadata operations from being included in the data transfer results, we build our own I/O benchmark upon MPI and perform simple sequential read and write on the given size

```
<Time Stamp (seconds)> <Duration (seconds)> <pid> <Open(o),Close(c),Write(w),Read(r)> <File path> <Offset(bytes)> <Size(bytes)>
1478024308.996711       0.000462            25914      o                              /path/to/file      0              0
1478024308.998007       0.000860            25914      r                              /path/to/file      0              16384
1478024308.998506       0.000017            25914      r                              /path/to/file      16384          12174
```

Figure 5.6: An example of trace files

for multiple iterations. We use this specially-designed benchmark to evaluate the I/O throughput of the file system.

5.3.4 Distributed I/O Trace

As a trace-driven simulation, we know our target applications from their I/O traces. We use MUSE [75] to generate trace files from the executions of a set of distributed applications. MUSE is an I/O tracer implemented on top of FUSE [36] and allows tracing of the I/O operations in arbitrary kinds of applications that run on Linux. In the trace, five important aspects of each I/O operation are recorded:

- **timestamp**: timestamp of the start time of each operation.
- **operation**: types of each operation – write, read, open, close.
- **filepath**: relative path of the file on which operations are performed. This is used as the identifier of file.
- **offset**: offset in the file where each operation starts.
- **size**: size of each operation performs.

We trace each application in a distributed way: First we mount MUSE on all the nodes used. Then we run the application in parallel, and MUSE traces the I/O on each node. After the execution, we merge the trace files from all nodes into a single file by sorting the timestamp of the operations. Since we lack exact clock synchronization during the execution, there are clock drifts among the timestamps from different nodes. However, since we run the applications in parallel, the exact order of I/O operations is nondeterministic, hence we consider the clock drifts as a part of the nondeterminism.

FUSE Overhead

FUSE-based I/O trace tools have their downside: FUSE introduces a significant amount of overhead, which reflected as the application's performance in the trace

Table 5.1: Experiment Environment

System	TSUBAME KFC
CPU	Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
Memory	62 GiB
Network	Mellanox FDR InfiniBand HCA
Storage	NFS
Nodes used	38

file. Moreover, the overhead of FUSE involves complicated context switches and memory copies, which can hardly be modeled [81]. Such ghost overheads significantly reduces the accuracy of the simulation.

In order to measure the FUSE overhead, we created an empty FUSE filesystem: EmptyFUSE. No actual I/O takes place in EmptyFUSE, only an in memory structure is maintained to record the file metadata, e.g. file size, of each file. Since there is no actual I/O overhead, the I/O delay from EmptyFUSE represents the FUSE overhead. Similar to the system benchmark, different transfer size would introduce different FUSE overhead, hence we measure the FUSE overheads on multiple transfer sizes. We use the benchmark results of the EmptyFUSE to adjust the timestamp in the trace file and remove the FUSE overhead from the trace file with following process.

$$newTS_i = newTS_{i-1} + IOthinkingTime - FUSETime \tag{5.6}$$

$$IOthinkingTime = originalTS_i - originalTS_{i-1} \tag{5.7}$$

$$FUSETime = \frac{size}{FUSEresults} \tag{5.8}$$

where the $newTS_i$ is the new timestamp for a record in the trace, while the $originalTS_i$ is the original timestamp from the trace. $size$ is the corresponding field in the same record (Section 5.3.4), and $FUSEresults$ is the benchmark result with the transfer size closest to $size$.

5.4 Simulation Results

In order to answer the three questions in Section 1, we perform a group of simulations using five different real word applications with the I/O data details listed in Table 5.3. We conduct our experiments on the TSUBAME KFC supercomputer with the specifications in Table 5.1. We show three studies using the five applications. We first vary the buffer size and show how the performance

Table 5.2: Input Data Size and Total access size of Applications

Benchmarks	Input size (MB)	Total access size (MB)	Total I/O size (MB)
Montage	1,200	7,500	27,000
Povray	14	25	760
Similar Pages	54	63	117
Miranda_IO	No Input Data	444*N	888*N
BTIO	No Input Data	6,800	13,624

Table 5.3: I/O patterns of Applications

Benchmarks	Scaling	I/O pattern
Montage	Strong	Mixed
Povray	Strong	Mixed
Similar Pages	Strong	Mixed
Miranda_IO	Weak	N-N
BTIO	Strong	N-1

changes with different buffer size (question i, Section 5.4.1). Secondly, we compare the performance ratio between the file-level swapping approach and the chunk-level swapping approach (question ii, Section 5.4.2). Thirdly and finally, we show the performance difference between four data replacement algorithms (question iii, Section 5.4.3).

As we mentioned in Section 5.3.3, the simulation starts with the system benchmarks. We utilize the NFS as the L2 storage, and an open-source software-level burst buffer system HuronFS [35] as the L1 burst buffer. We benchmarks the I/O throughput performance on both NFS and HuronFS for up to 32 nodes on the TSUBAME kFC. We configure the HuronFS to have one meta data server and one IO server, and keep the same configuration for all the experiments. We benchmarks the I/O throughput with our micro-benchmark and repeat each measurement for 50 times. We vary the I/O size from 1 KiB and increase by 1 KiB until 128 KiB, and then increase by 100 KiB until the 5 MiB, where the performance plateaus.

5.4.1 Simulating with Real Applications

We trace five different applications: Montage [14], Povray [70], Similar Pages, BTIO [61], and Miranda_IO [58] on TSUBAME supercomputer [20]. The first three applications: Montage, Povray, Similar Pages are workflow applications, consisting of multiple small jobs with mutual dependencies. The dependencies are described

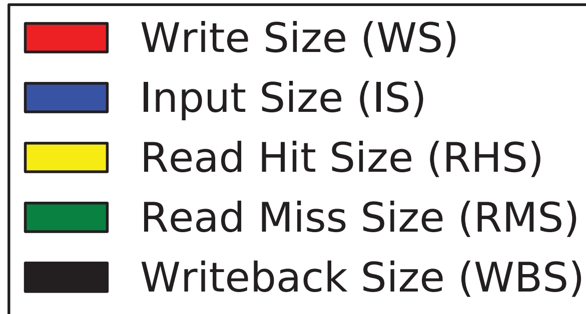


Figure 5.7: Performance Counters in Different Colors

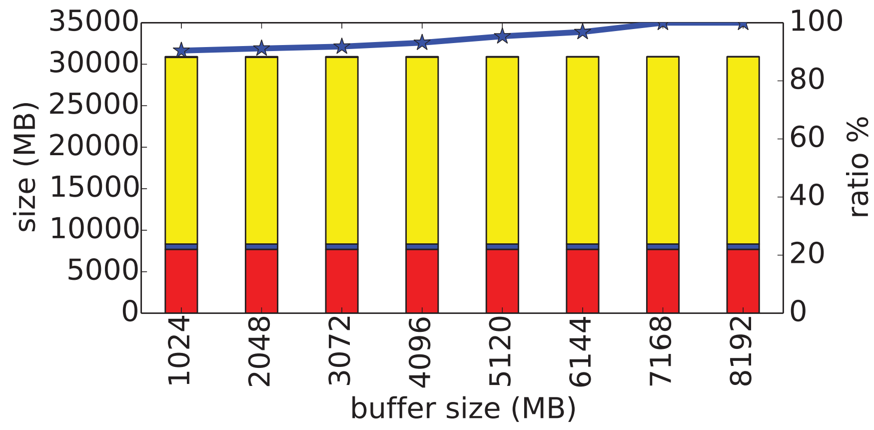
by using GNU Makefile, and we use a framework called gxp [7] to execute the "make" command in a distributed manner. The last two applications: BTIO and Miranda_IO are MPI applications. We trace all the applications with 32 compute nodes. We set the chunk size in the simulator to 1 MiB for Montage, BTIO, and Miranda_IO to avoid bad performance from frequent small I/Os. However, for the Povray and Similar Pages, which consist of thousands of small files, we use a small chunk size of 4 KiB to avoid waste of space caused by internal fragmentation. In Table 5.3, we show the total access space under the given chunk size of each application, which defines the minimum required capacity in current burst buffer strategy to buffer all the files. In the first experiments, we use the most common used algorithm, LRU, as the data replacement algorithm for the buffer.

In the figure we show the performance ratio to the theoretical performance according to Formula **??**. We use stacked columns to show the details of the performance according to the counters introduced in Section 5.3.1 with different colors.

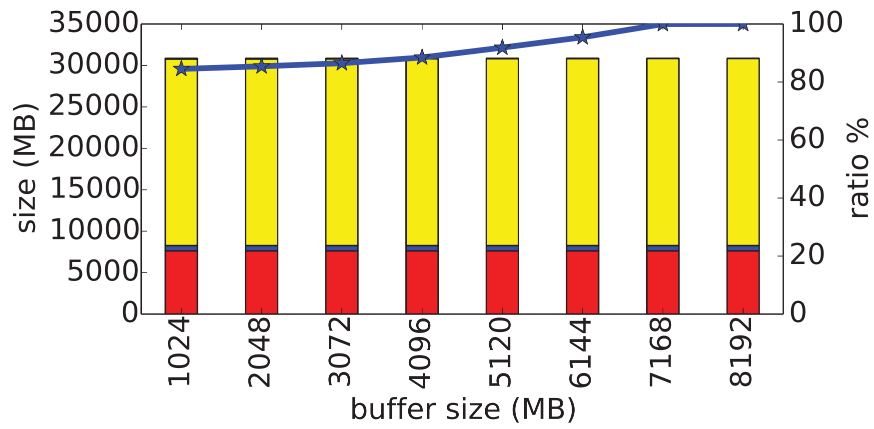
Montage

A workflow application developed in NASA that constructs custom science-grade mosaics by composing multiple astronomical images [14]. The whole workflow consists of three parallel phrases where the same operations are applied on different data and between them are several reduce phrases. We run the montage in a strong scaling manner.

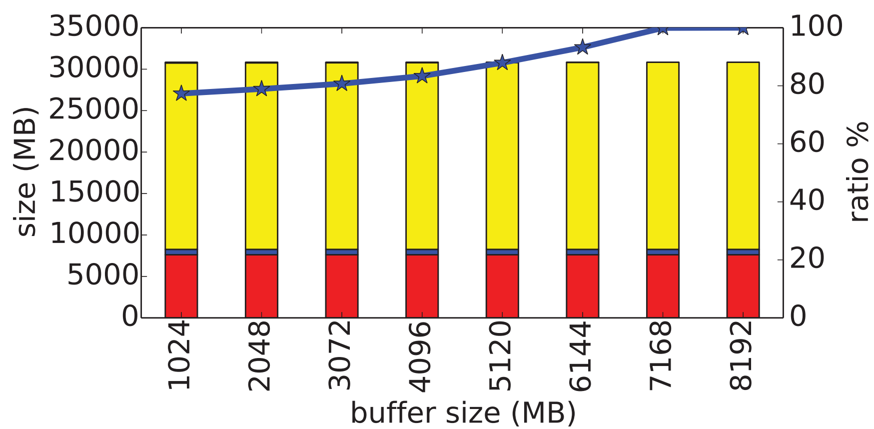
Figure 5.8 shows the simulation results of Montage. As we expected, the performance of the application increases when we have more buffer available. The performance counters on the stacked columns indicate such increment as the



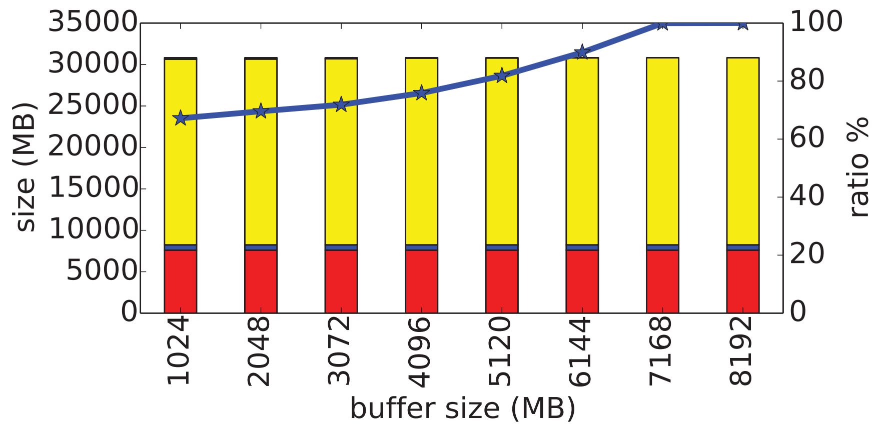
(a) 1 node



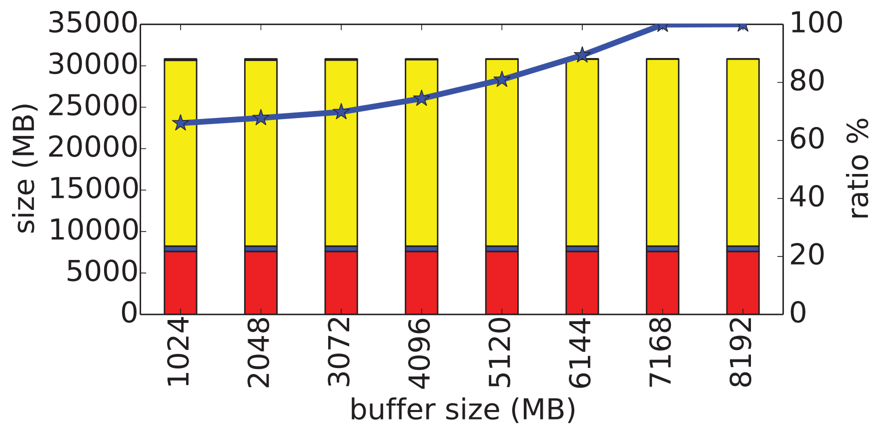
(b) 2 nodes



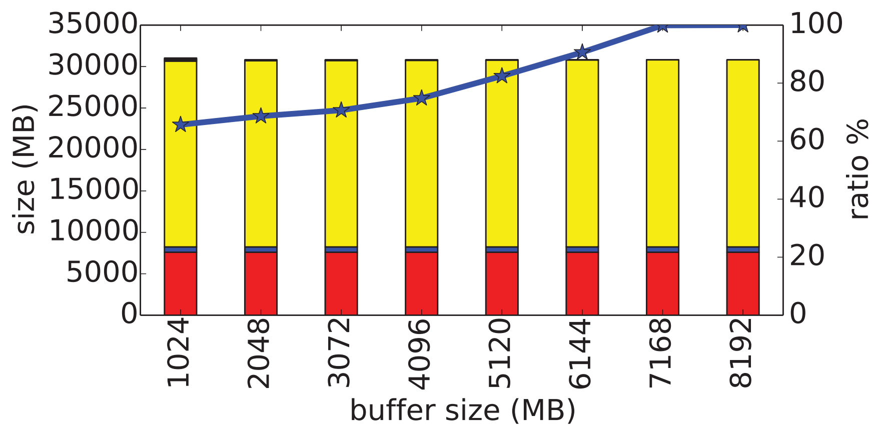
(c) 4 nodes



(d) 8 nodes



(e) 16 nodes



(f) 32 nodes

Figure 5.8: Simulation result of Montage

reduction of the writeback overhead on the swap out. With more buffer available, more dirty data can be buffered, and simultaneously updated on the background. However, with only a 6 GB burst buffer, which is about half of the total space Montage accesses, we can achieve over 90% of the theoretical max performance.

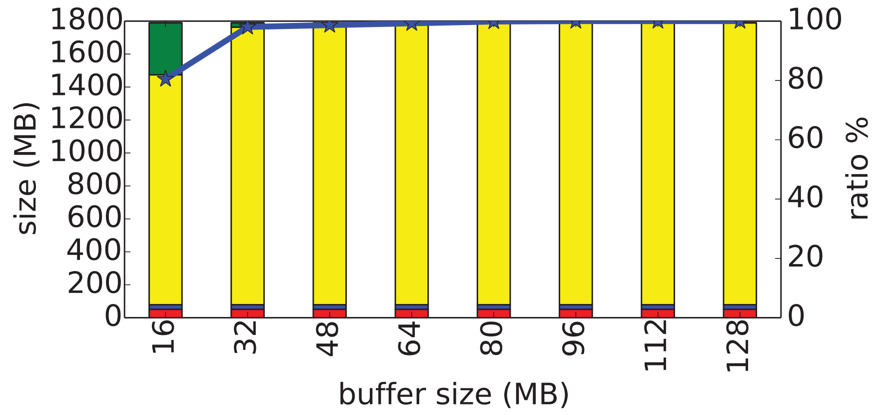
As shown in the Figure, when we scale up the application, the performance ratio reduces. That is because the I/O bandwidth increases when having more nodes, but the throughput of the L2 does not change as the number of IO server is fixed. Hence the application needs to achieve higher performance to keep the same ratio when scale up. Moreover, with more nodes running in parallel, more different data will be accessed at the same time, therefore, more buffer is needed to store the data and achieve the same performance. The drawback from the slow L2 is more significant when there are more swaps happen in the small buffer size case. However, even in the worst case with 32 nodes, by using only a 4 GB burst buffer, which is around half of the total size Montage accesses, we can achieve 70% of the theoretical max performance.

Similar Pages

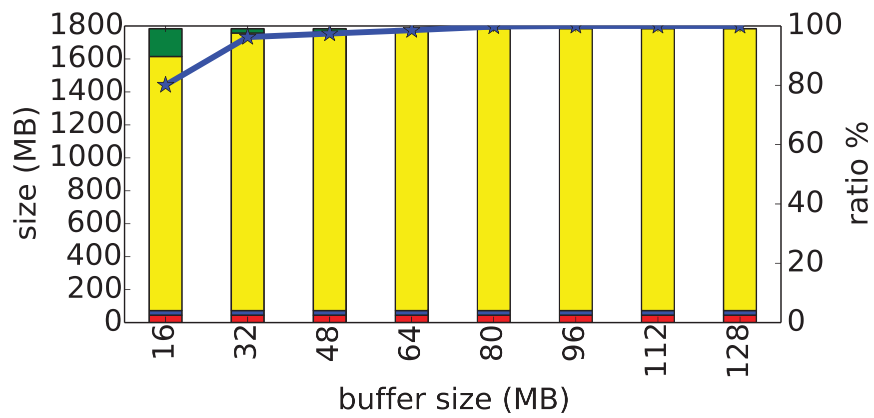
A text processing application. Similar pages reads a large amount of text taken from webpages and analyses all the pairs of similar sentences which have smaller Hamming distance than a certain value.

As shown in Figure 5.8, the main operation is reading, which has a very different access pattern compared to Montage. As the figure suggests, Similar pages is able to achieve extremely high performance with limited buffer. With 32 MB buffer, we can again achieve over 90% while the total access space for the Similar Pages is 94 MB. This is because Similar Pages uses a typical $O(N^2)$ all-to-all comparison, where each file needs to be reused N times. When running in parallel, all the jobs read the same files and conduct comparison simultaneously, so data locality is extremely high in Similar Pages and only a small buffer size is required to achieve high performance.

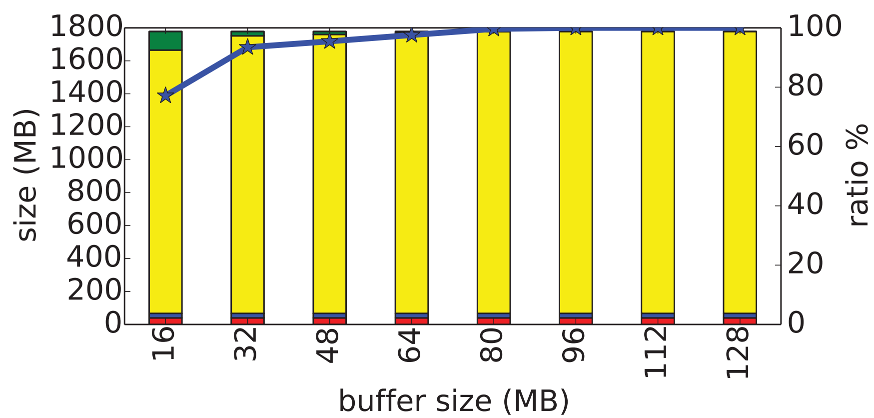
We observe the same scaling trend as with Montage, but with far less buffer to achieve high performance. The performance ratio of 16 MB buffer drops from 80% with 1 node to 60% with 32 nodes. Thanks to the high data locality of Similar pages, we are still able to achieve around 90% of the theoretical maximum performance with 32 MB in all the cases.



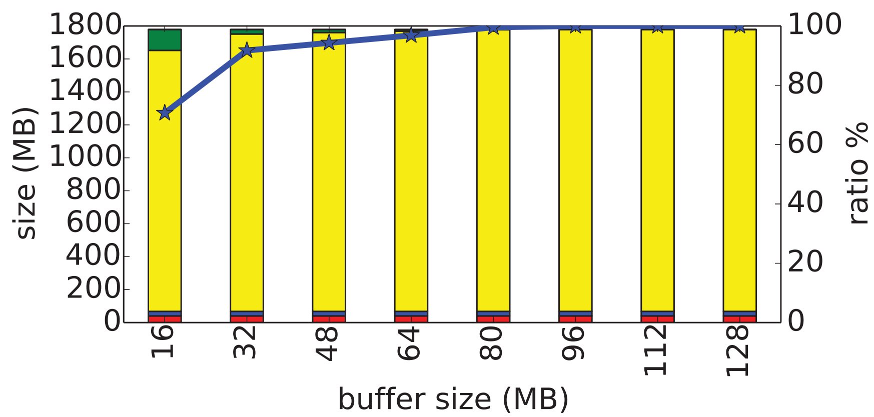
(a) 1 node



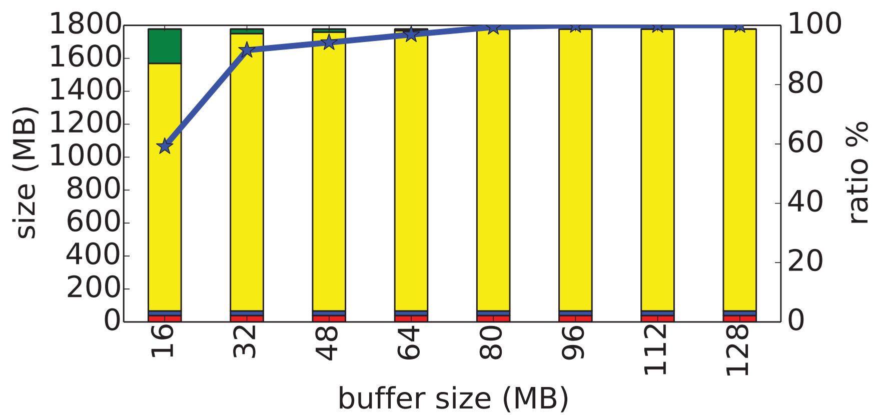
(b) 2 nodes



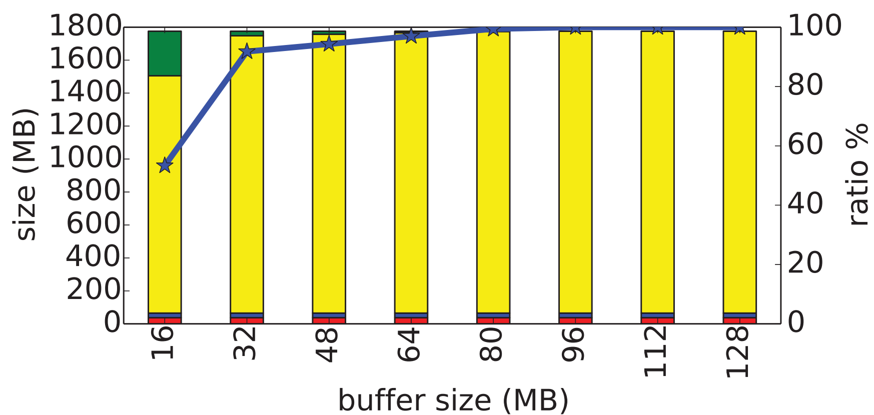
(c) 4 nodes



(d) 8 nodes

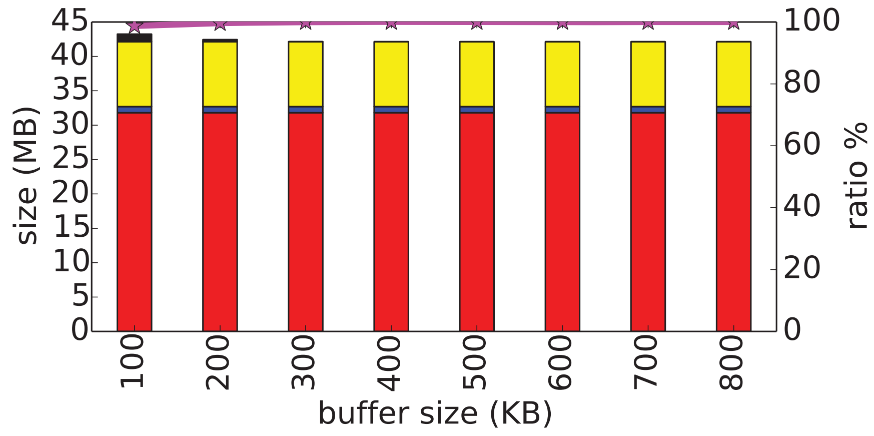


(e) 16 nodes

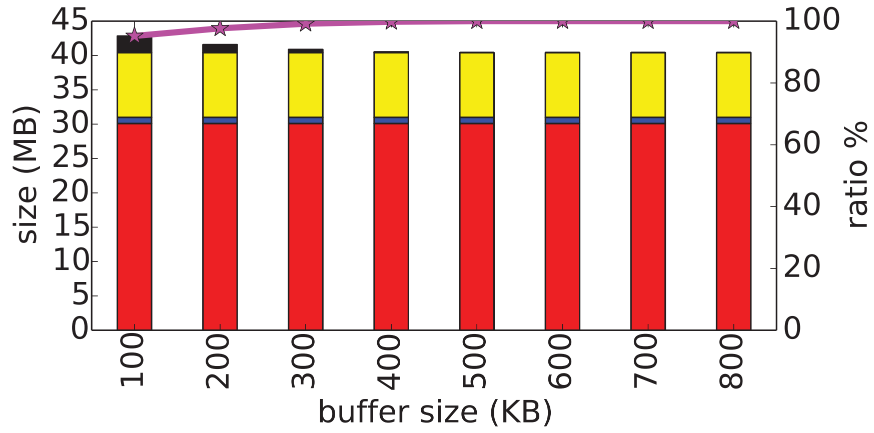


(f) 32 nodes

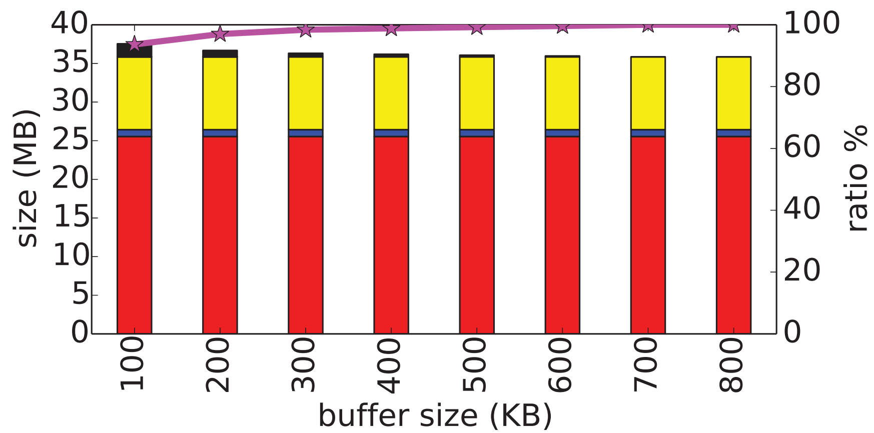
Figure 5.9: Simulation result of Similar Pages



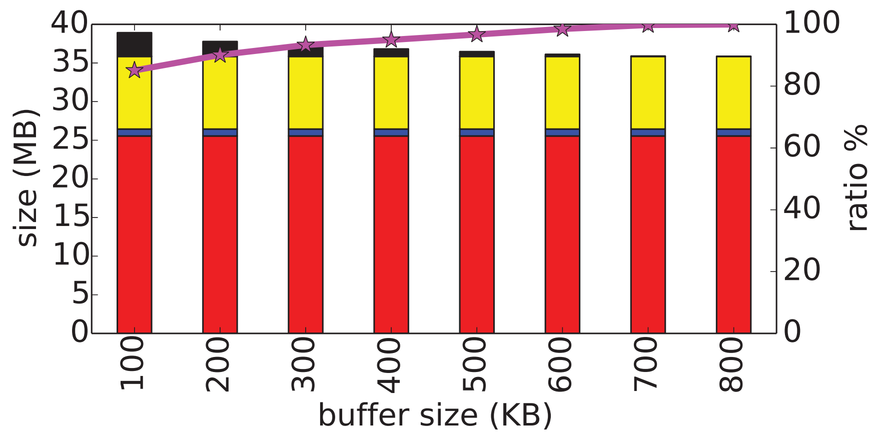
(a) 1 node



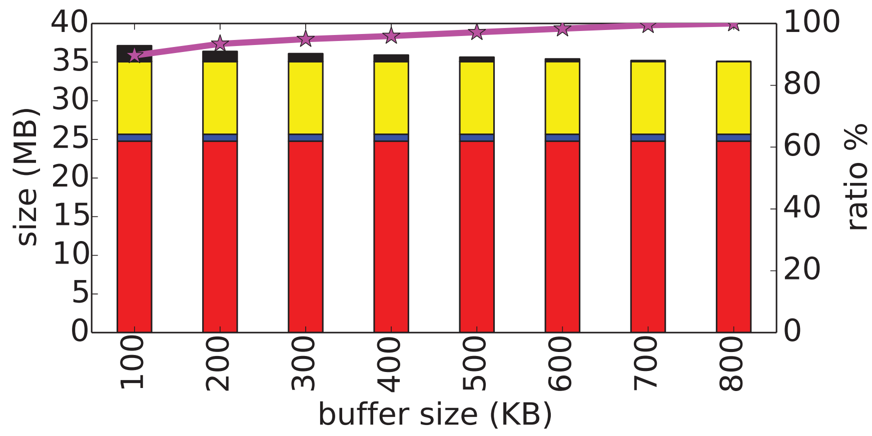
(b) 2 nodes



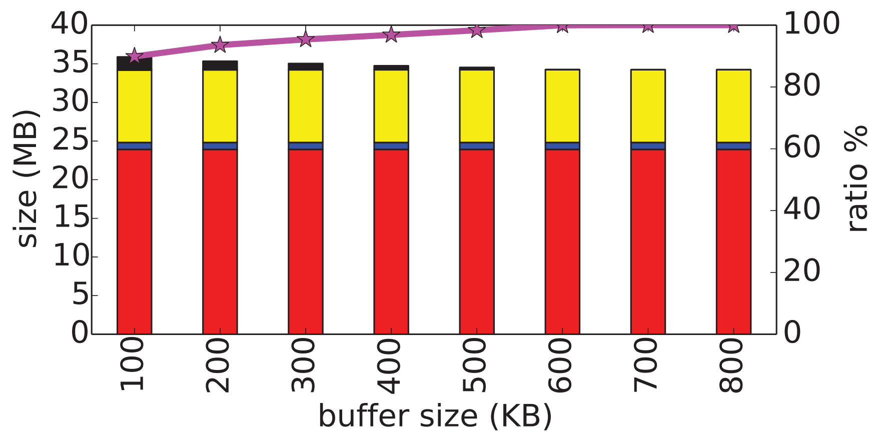
(c) 4 nodes



(d) 8 nodes



(e) 16 nodes



(f) 32 nodes

Figure 5.10: Simulation result of Povray

Povray

A software toolkit for creating three-dimensional graphics using Ray-tracing [70]. Povray generates an image from a collection of source files describing objects and lights in a scene.

Figure 5.10 shows the results for Povray. In Povray, the input is source files. Similar to language compilers, Povray shows extremely high data locality: each source file is only accessed when process its image. During the process, each file is read multiple times. Due to such high data locality, Povray is able to achieve almost full performance with extremely limited buffer capacity (hundreds of KB) compared to its total access space (few MB).

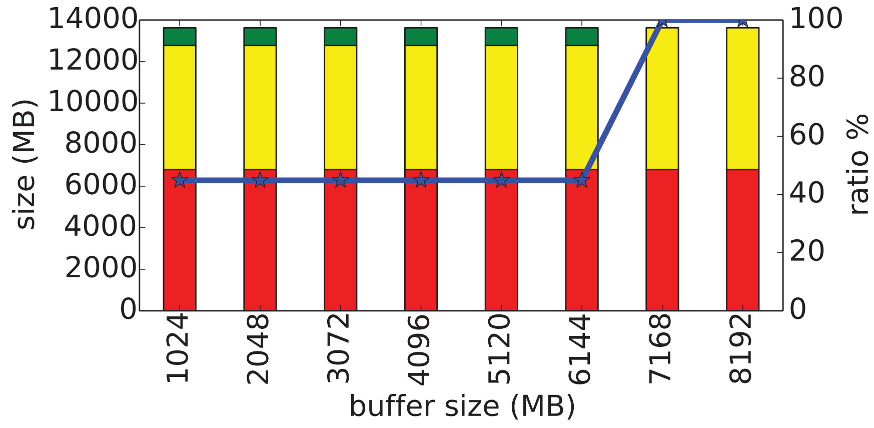
BTIO

BTIO [61] is an I/O benchmark from the NAS Parallel Benchmark suite. BTIO solves Block-Tridiagonal (BT) NPB problem, and implements using MPI with a fairly complex domain decomposition called diagonal multi-partitioning. We run BTIO in MPI-full mode, in which all the processes share the same file. During the execution, all the processes read and write to different offsets of the same file at the same time, hence, the entire file is accessed in each iteration. BTIO writes for 200 iterations with all the processes, and again reads for 200 iterations.

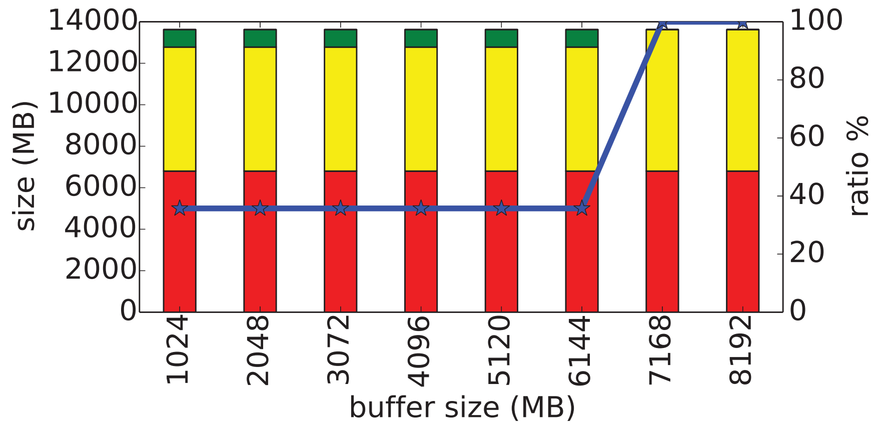
As Figure 5.11 indicates, BTIO is totally different from the previous applications. Very limited performance improvement is achieved by having more buffer. However, after it reaches the total access space, the performance ratio jumps to 100 % immediately. As we expected, with the entire file being accessed at all the time, frequent data swapping happens between the L1 and L2 storage if we do not have enough buffer, which cause a significant performance degradation.

Miranda_IO

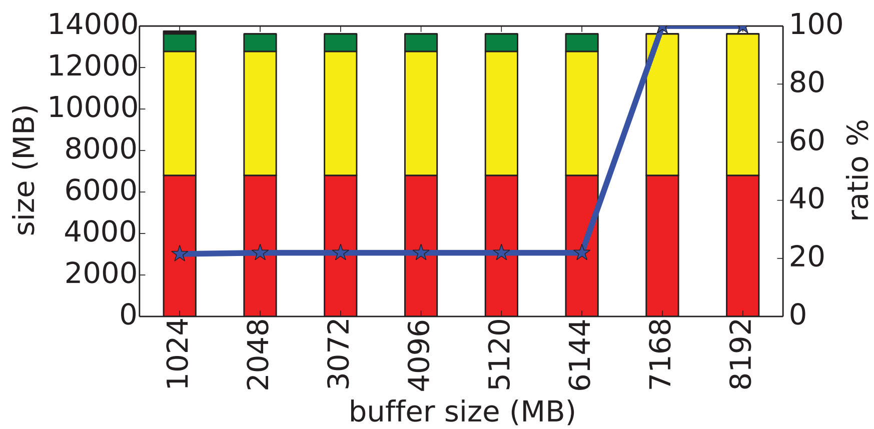
Miranda_IO [58] is the I/O kernel from the Miranda hydrodynamics application code, implemented in MPI. The I/O pattern of Miranda_IO is close to checkpointing: it runs for several iterations; in each iteration, each process overwrites data to its own file, which has a size of 111MB; after finishing writing, each process starts reading the output files from its neighbor processes; then, moves on to the next iteration. In this experiment, we run Miranda_IO with 4 processes per node for 1 iterations, to perform a checkpoint-like I/O pattern, and the total I/O size is $444MB \times 2 \times N$.



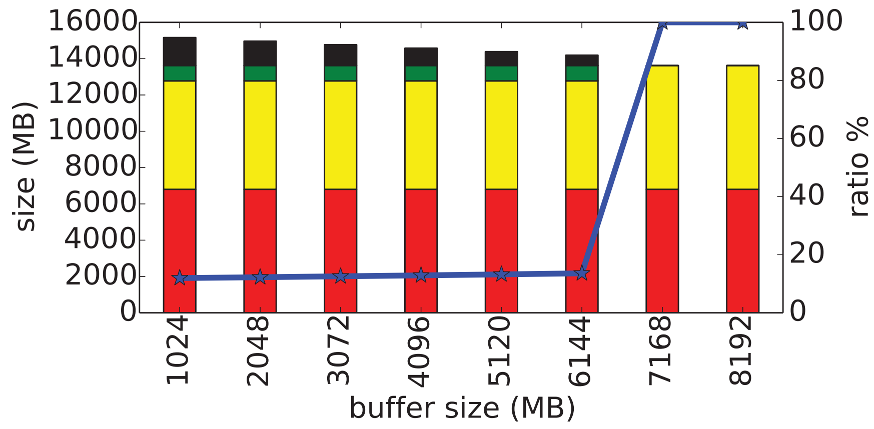
(a) 1 node



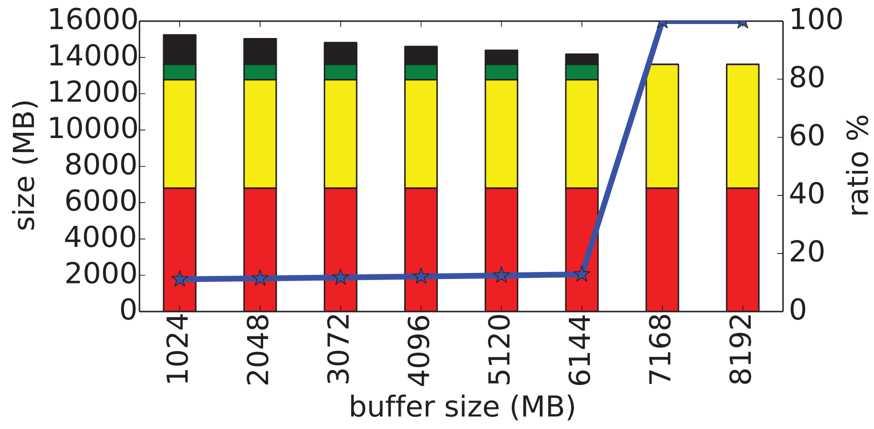
(b) 2 nodes



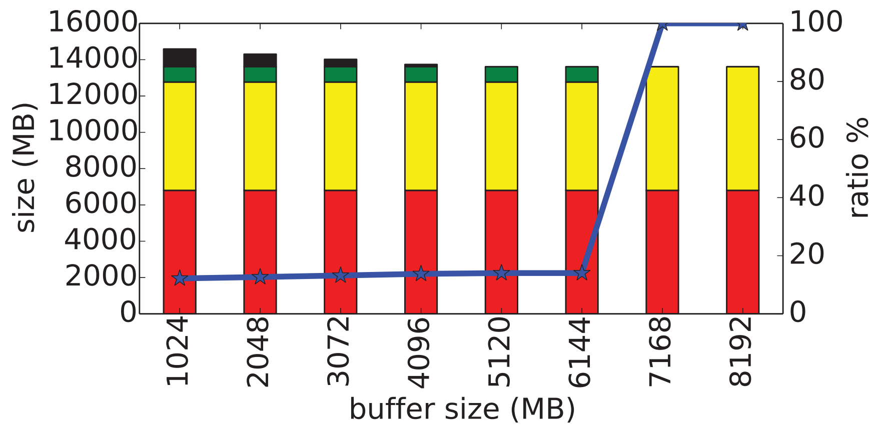
(c) 4 nodes



(d) 8 nodes

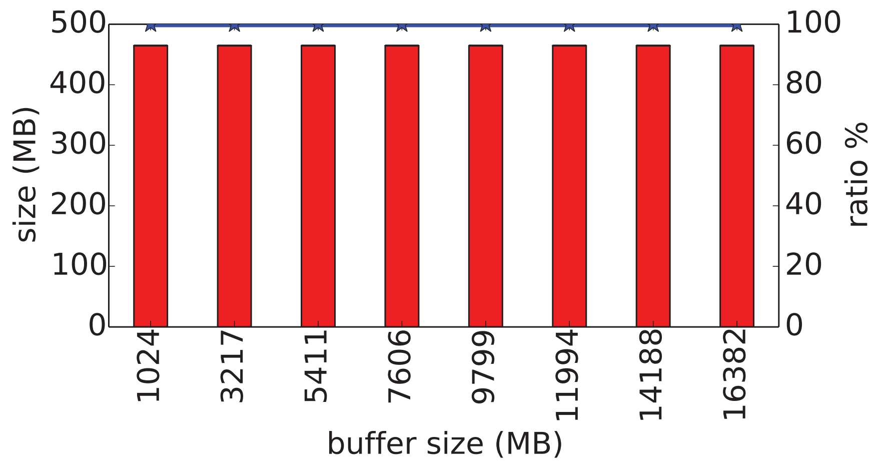


(e) 16 nodes

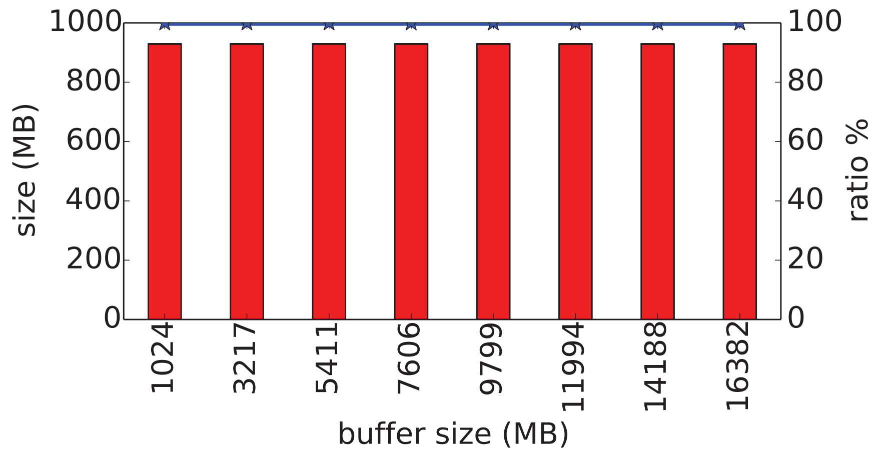


(f) 32 nodes

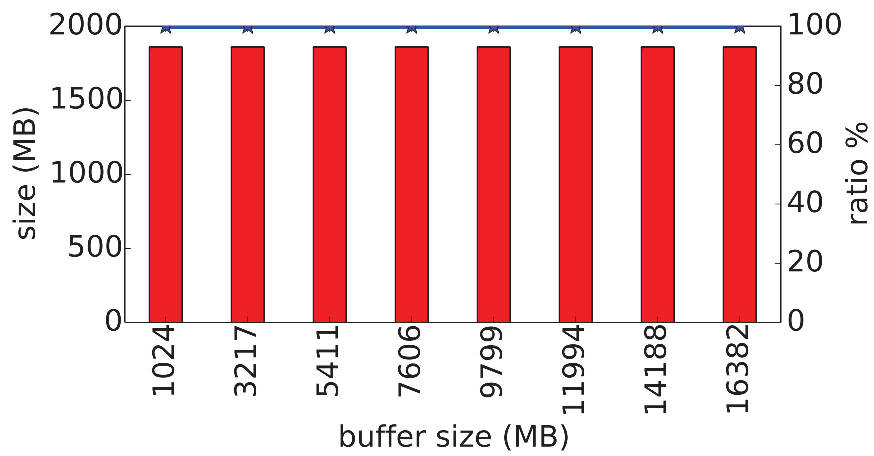
Figure 5.11: Simulation result of BTIO



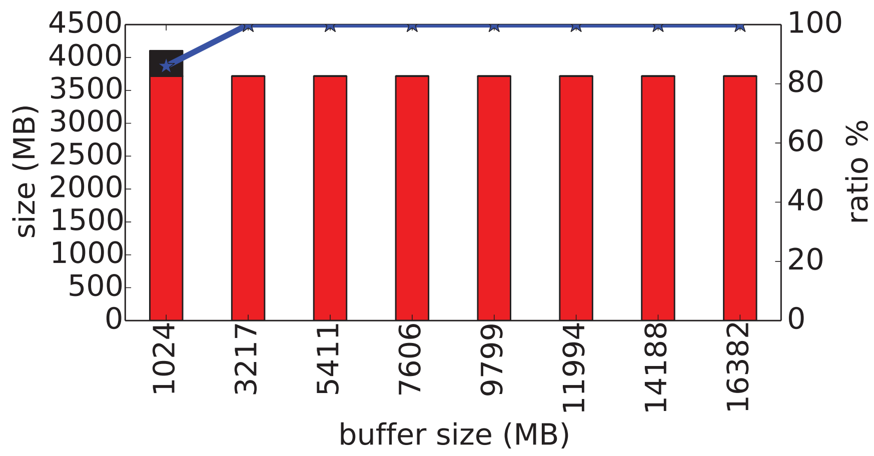
(a) 1 node



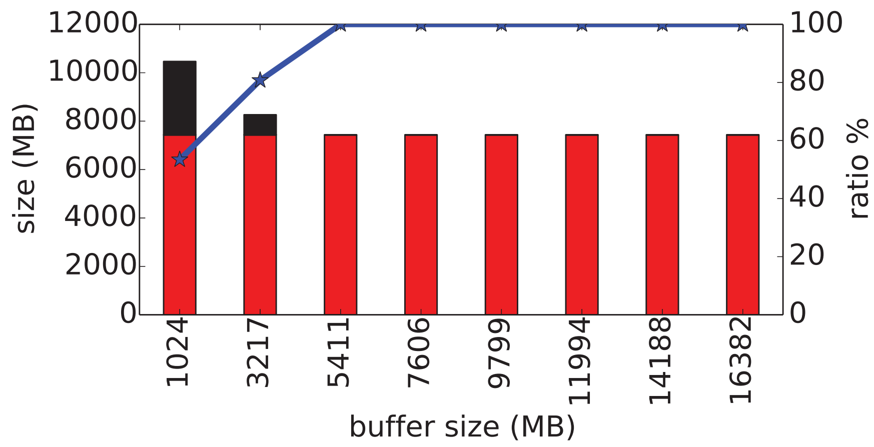
(b) 2 nodes



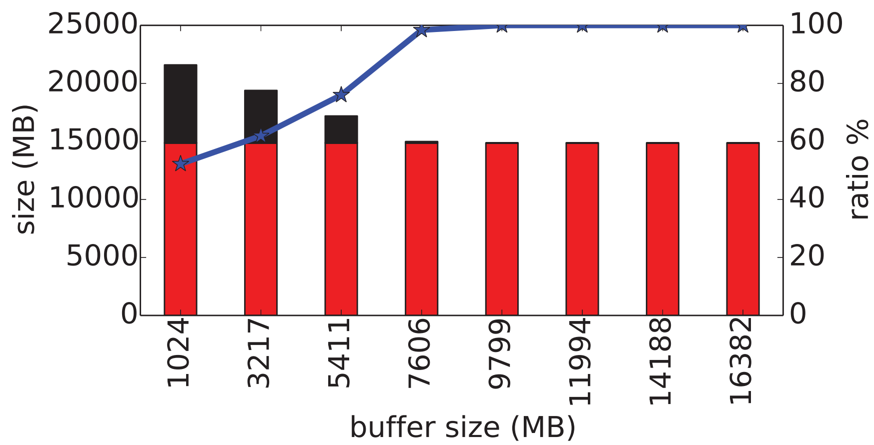
(c) 4 nodes



(d) 8 nodes



(e) 16 nodes



(f) 32 nodes

Figure 5.12: Simulation result of Miranda_IO

The access patterns of Miranda_IO is similar to the BTIO, but Miranda_IO only performs writes. However, unlike the BTIO, in most of the cases, we can see that even in all writes applications with less iterations, performance gets improved by having larger buffers, since a larger buffer can cache more data before the buffer fulls.

From these experiments, we found that for common applications, having less buffer than the application access space can still provides high performance. The higher data locality it has, the less buffer it needs. However, with data parallel, high parallelism can lower the data locality. For checkpoint-like applications with the high number of iterations, the buffer size should be large enough to fit the entire problem size in order to achieve good performance.

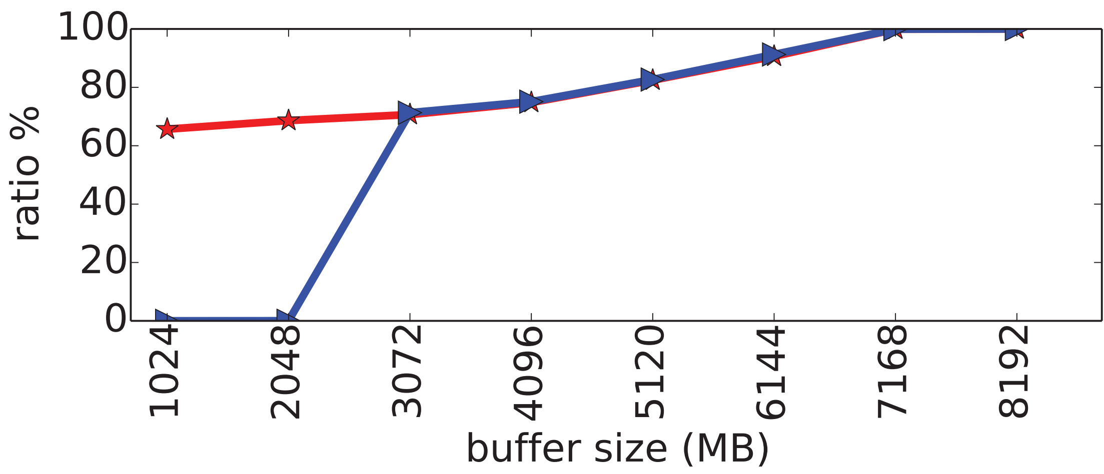
5.4.2 Chunk-level VS. File-level

In the second experiment, we compare the performance of chunk-level swap-in/out and file-level swap-in/out to demonstrate the benefits of having chunk-level swap-in/out. Due to the space limitation, we show only the 32 node cases for all five applications. Note that cases with less than 32 nodes shared the same performance trends as the 32 node case.

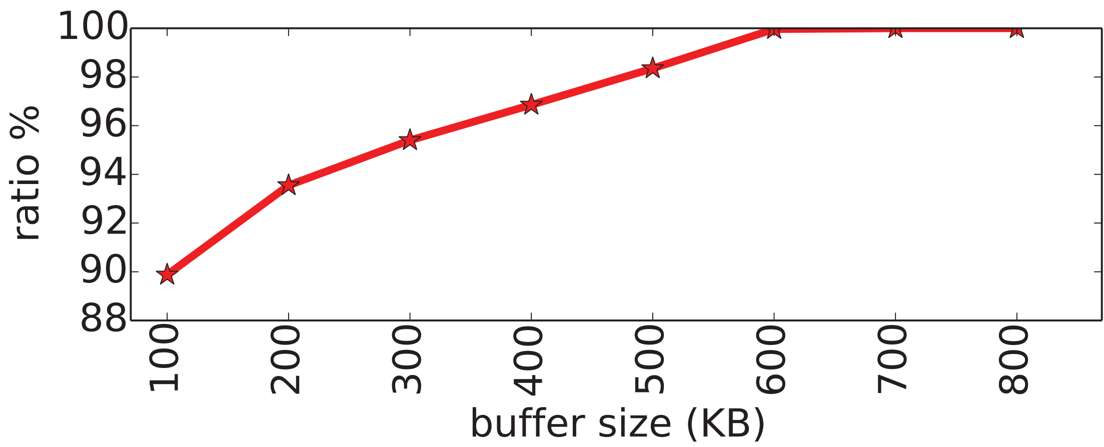
Figure 5.13 shows the performance comparison. As we can see in all the cases, the higher granularity chunk-level swap-in/out achieves higher performance than file-level swap-in/out. When the file are small (Similar Pages) or close to the chunk size (Montage), the performance difference are slight. With more data to swap in/out, file-level pays more performance penalties on swap-in/out due to unused parts of the file. Moreover, file-level swap-in/out can not support any buffer size that is smaller than any of the files, the missing points in BTIO, since at least one of the file cannot fit into the buffer. However, chunk-level swap-in/out can support such cases since the chunk size is smaller than the file size. Hence, from this experiments, we show that chunk-level swap-in/out has advantages over file-level swap-in/out on both performance and usability.

5.4.3 Simulation with different data replacement algorithms

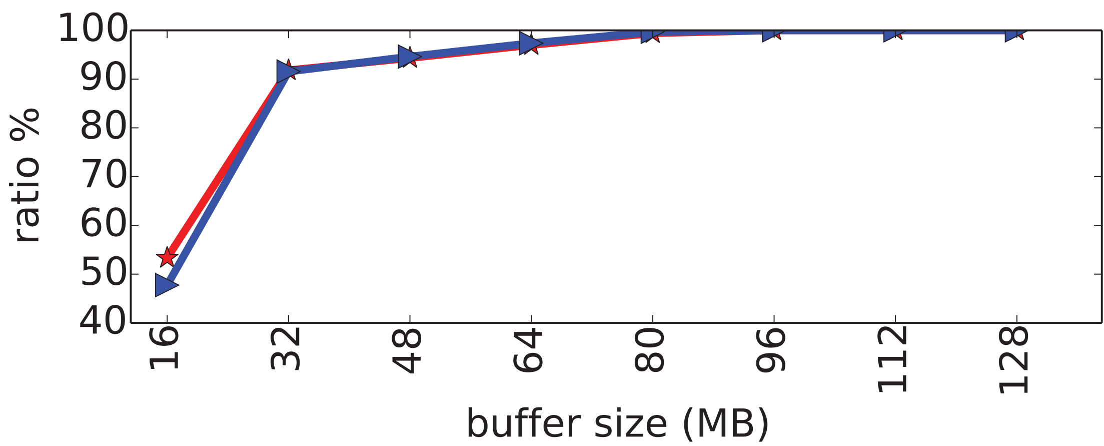
As we discussed previously, besides the different burst buffer size, the impact of different data replacement algorithms on the performance of applications is also unclear. Hence, we simulate with following five common data replacement algorithms to demonstrate the impacts.



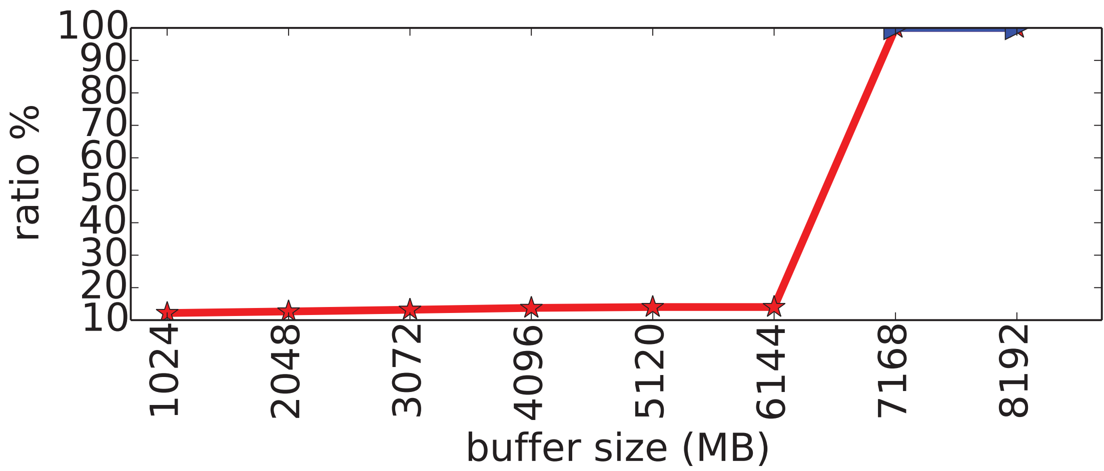
(a) Montage



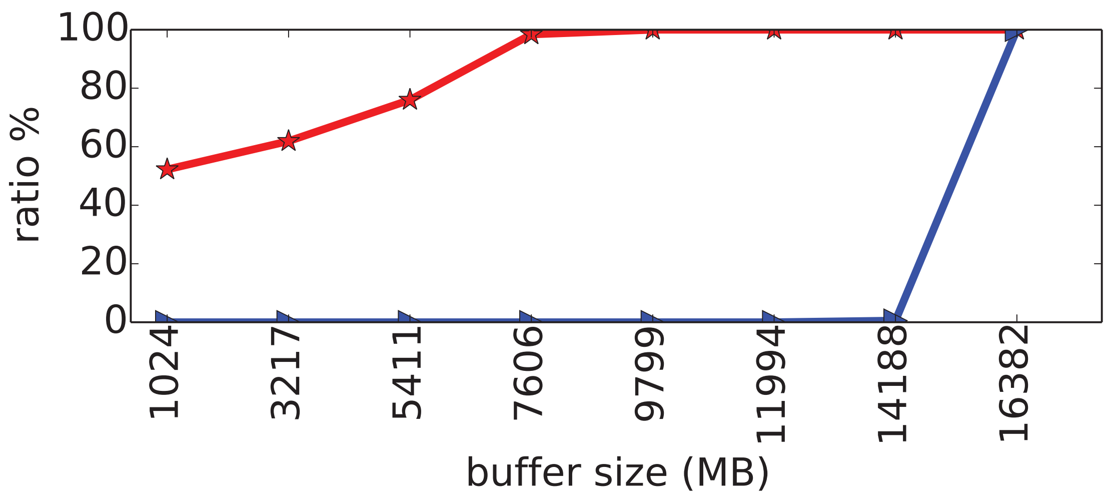
(b) Povray



(c) Similar Pages



(d) BTIO



(e) Miranda_IO

Figure 5.13: Simulation Result With Chunk-level and File-level swap-in/out

- First In First Out (FIFO) : FIFO is the simplest data replacement algorithm. It places the data blocks in the order of its first appearance. The block that appears first will be swapped out first.

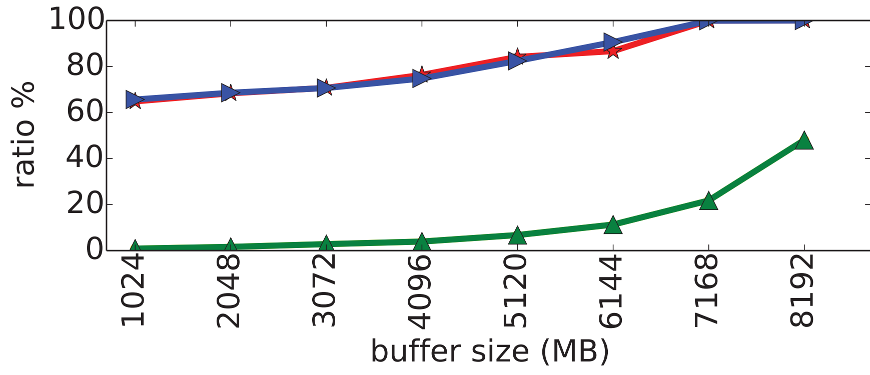
- Least Recently Used (LRU): LRU is the most common data replacement algorithm. LRU keeps track the used time of each chunk and swaps out the least recently used one. The design of LRU follows a very important phenomena in data access, i.e. data access locality, to maximize the cache hit rate. LRU has been implemented and used in many cache/buffer systems.

- Least Frequently Used (LFU): similar to LRU, LFU swaps out the least frequently accessed blocks. The design is based on an idea that the more a chunk was accessed, the more popular it is, and the more probable it would be accessed again in near future. However, in many cases, LFU suffers from buffer pollution, where some blocks get accessed too many times in the past, and ended up staying in the buffer forever.

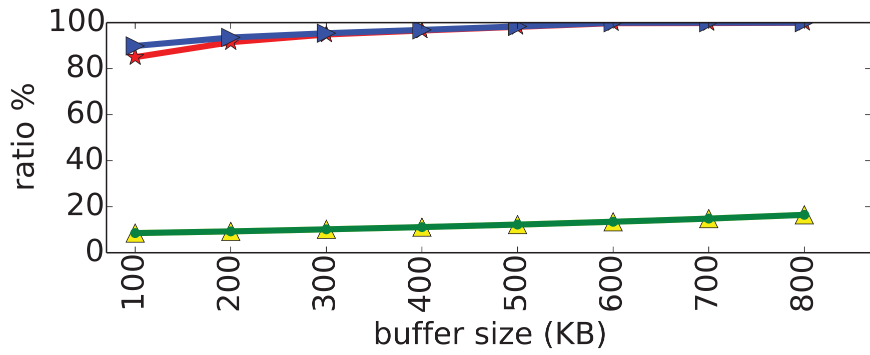
- Least Frequently Used with Dynamic Aging (LFUDA) : LFUDA was proposed to solve the cache pollution issues in the LFU. LFUDA adds an aging factor to the counter in LFU, so that new blocks can quickly surpass the old blocks with a high counter value, and then evict the old blocks out of the cache.

We simulate all the applications studied in Section 5.4 with the five data replacement algorithms for the 32 node cases. From Figure 5.14, we see significant performance differences between algorithms. LRU and FIFO can achieve good performance for almost all the applications tested. Despite its simplicity, FIFO also achieves very good performance. LRU and FIFO only perform slightly worse than others for BTIO when the whole file is being accessed at the same time, which results in a random-access-like access pattern with very limited data locality. Such an access pattern breaks the assumption of LRU. On the other hands, LFU and LFUDA performs relatively poorly in most of the cases, especially for the Montage case. They two algorithms suffer lot due to the buffer pollution.

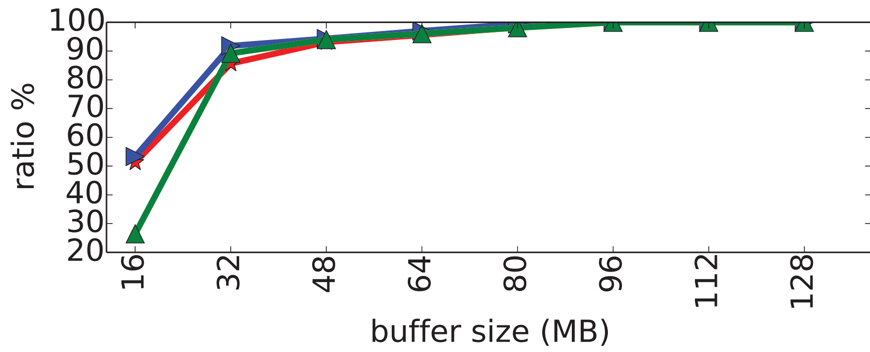
In this experiment, we see a huge performance gap between different algorithms. It is also clear that certain algorithms are more suited for certain applications due to the assumptions of the algorithm matching the application's access pattern.. We conclude that a wise choice of data replacement algorithm will have a huge impact on the performance on burst buffer.



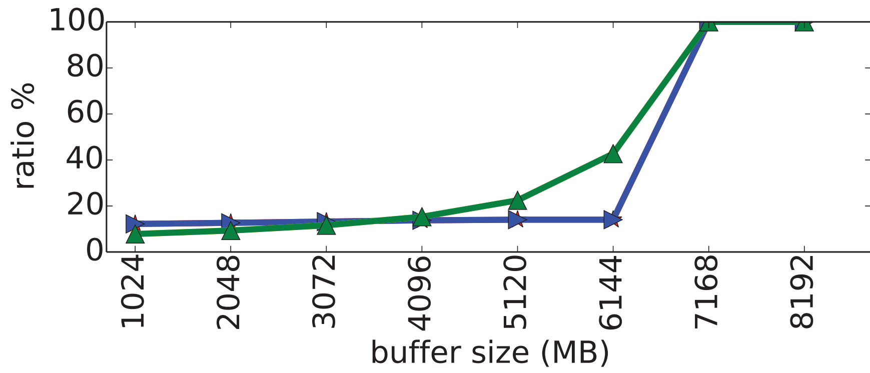
(a) Montage



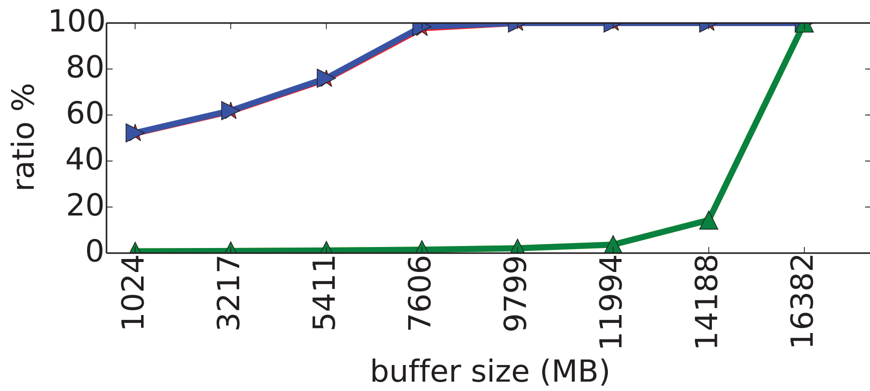
(b) Povray



(c) Similar Pages



(d) BTIO



(e) Miranda_IO

Figure 5.14: Simulation Result With Different Data Replacement Algorithms

5.4.4 Discussion

In order to maintain the generality of our burst buffer model, we do not include detailed implementation features into our model. Several implementation details or optimizations can significantly impact performance, which can cause the mismatch of the predictions. For instances, we expect that :

- In a real file system, locks can be implemented to guarantee the correctness and consistency of the data, which can limit the asynchronous write back.
- Client side buffer are introduced to group multiple small I/O operations into larger, consolidated operation to improve the performance, which can change the I/O patterns from the viewpoint of the filesystem.

- Buffered I/O or pre-staging are performed on L1 storage for high performance, which can improve the L2 read/write performance and reduce the access time.

Such features highly depend on the different implementations and are not portable enough to be included in our model. Since matching the actual run time of application on a given burst buffer system is not the focus of this paper, we demonstrate how how variations in configurations can produce consistent performance trends for different application types. We argue that the trend of the performance with different buffer size or algorithms on the same system would not be significantly affected the implementation details and optimizations based on our findings. Future work can include a more complicated burst buffer model and comparative analyses to support the design of future I/O architectures.

5.5 Simulation with Parallel Execution

As we described, our simulation is a trace-based simulation. The simulation takes the I/O trace from execution to understand the I/O patterns of the applications. The simulation is build upon the assumption that the I/O patterns would not be significantly different across different runs, which holds for single-threaded applications. If the traces are different, the simulation results can also be different. The applications in HPC are built with multi-threads and run on hundreds and thousands of nodes for extreme high performance, for such applications the execution can be non-deterministic due to the parallelism and can be different across the runs. Such parallel execution pattern breaks the foundation of the trace-based simulations, and might affect the accuracy and the conclusion we get from our burst buffer simulation.

In order to figure out whether our burst buffer simulator can be used for parallel applications, we conduct an analysis to answer the following three questions both theoretically and empirically.

- What execution patterns can affect the accuracy simulation.
- Do real applications contain such patterns.
- How significant is the difference.

In this section we give a detailed explanation about how the parallel execution might change the simulation results in section 5.5.1. Then we give hypotheses in

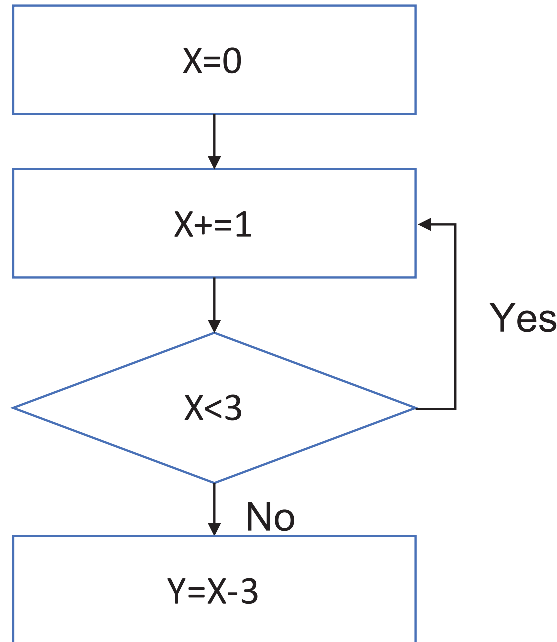


Figure 5.15: An example of a control flow

sections 5.5.2, and finally we test our hypotheses with real runs of applications on supercomputers in sections 5.5.3.

5.5.1 Execution Patterns Changed by Parallelism

Applications contains a defined control flow (Figure 5.15) by the programmer, which generates the binary files of the applications. On the executions, the CPUs interpret the binaries and execute with the given parameters, each thread generates a execution sequence (Figure 5.16) based on the binaries. The trace are the records of the execution sequence. The control flow is deterministic, however, the execution sequences can be different across different runs. Many things changes how the commands are actually executed, such as input parameters. For example, a for loop can be executed for only once or millions depends on the loop parameters, which generates two totally different instances.

In the single-threaded program, there is only one thread that executes the control flow. With given control flows and input parameters, the execution sequences are usually deterministic. Exceptions include programs involve random numbers or time based applications. However, in case of multi-threaded programs and multi-node programs, there can be thousands and millions of threads run the control flows. Each thread can have different paces on the executions and the speed

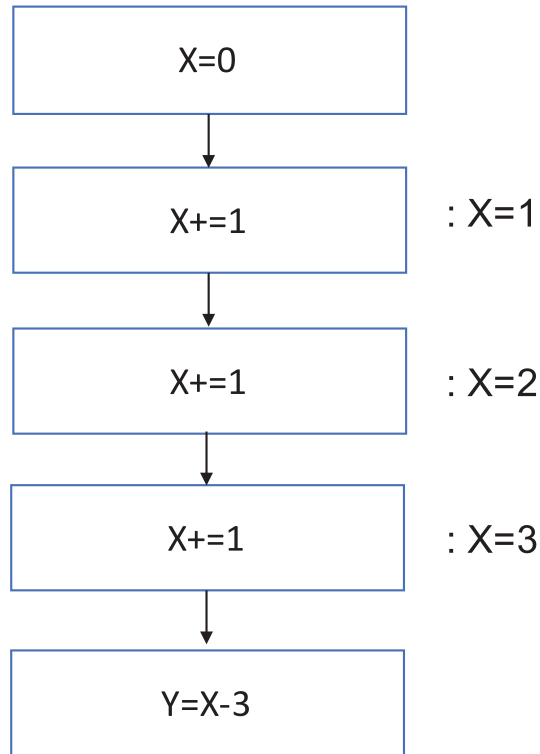


Figure 5.16: An example of a execution sequence

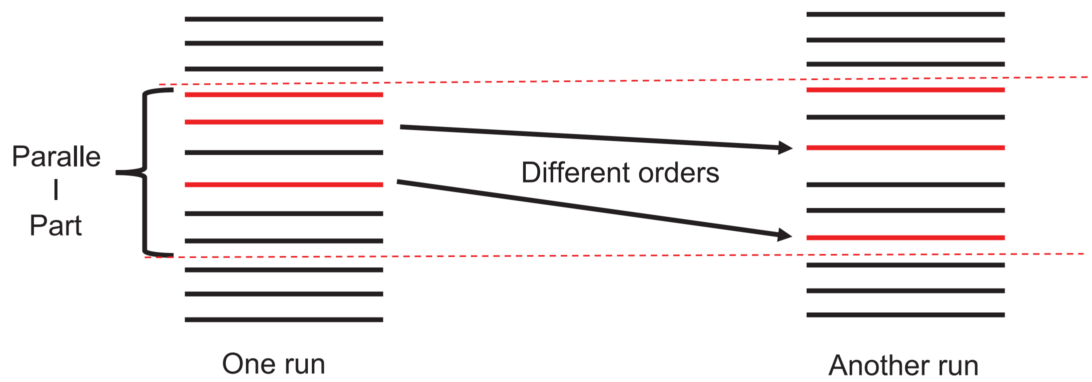


Figure 5.17: Execution pattern changes due to parallelism

of execution can be changed across executions. With different threads run in parallel, the order of commands can be different and generate different overall execution sequences. Synchronizations are used to synchronize the pace of multiple threads, and make sure the order of commands are deterministic. However, synchronizations introduce overheads. Moreover, synchronizations force the fast threads to stop and wait for others to finish, hence, the overall execution speed can be reduce. In order to achieve the maximum performance with multi-threading, programmers needs to remove the unnecessary synchronizations as many as possible and maximize the parallelism. Hence the execution sequences for large scale HPC applications usually contains large non-deterministic parts.

Since trace-driven simulation relies on the record of the execution sequence to reflect the control flows, such non-determinism can also affect the accuracy of trace-driven simulations including our simulation [63]. In order to guarantee the accuracy usability of our simulation, we need to conduct a further study on the impact of the non-determinism and the scope of our simulation.

5.5.2 Hypotheses

We first perform a theoretical analysis on the non-determinism of the trace of parallel applications, to answer the first two questions, and understand the impacts from the non-determinism and the application behavior that generates the trace that have huge impacts on the simulations. Here, we focus on the impacts on the buffer miss/hit rate and the final conclusion we have from the simulator, the absolute order of the executions will always change due to the parallelism. However, we want to see whether such change would impact the conclusion of the simulation.

First of all, with a buffer that can accommodate all the data that an application accesses, the non-determinism would not significantly change the conclusion as all the data will always be read from the parallel file system into burst buffer first, and stay in buffer for the whole run. As long as the commands stay the same, the changes of the order of the commands would not change the results of the burst buffer simulation.

If we do not have enough buffer to accommodate all the data, the situation becomes complicated. As we explained in the previous section, there are two major ways of execution in the parallel applications: in parallel and sequentially. The real applications are mixes of the two ways of executions. Totally parallel applications have no relationship between each thread, so they can be simply considered as multiple individual applications. On the other hand, totally sequential applications

are not parallel applications, as the commands are executed one by one. Moreover, there is no adjacent sequential parts without parallel part in between and no adjacent parallel parts without sequential part in between, otherwise, the two sequential/parallel parts can be consider as one. Hence, here we consider applications have following characteristics:

1. Containing multiple of parallel part and sequential parts.
2. There is a parallel part between two sequential parts, and there is a sequential part between two parallel parts.
3. No dependency inside each parallel part.
4. Dependencies exist between two parallel parts or between parallel and sequential parts.

With the defined characteristics of the applications, we found that the non-determinism caused by parallelism would not significantly affects the numbers of buffer miss/hit and the results of the simulation. We discuss in two situations:

- If we have big enough buffer in section 5.5.2.
- If we do not have big enough buffer in section 5.5.2.

Parallel Execution is not Totally Random Order

According to the characteristics 5.5.2, there are multiple parallel and sequential parts inside the applications. As shown in Figure 5.18, we have a red parallel part and a green parallel part separated by a black sequential parts. As we discussed, for such applications, the execution orders can be different across different runs. Although, the execution orders seem to be random, there is some patterns we are sure that would not appear in the execution orders due to the dependencies. For example, if we consider the execution order in Figure 5.18, the order can be different as the (1) and the (3) as shown in Figure 5.19, however, the (2) is impossible due to the dependencies between parallel part I and parallel part II. Hence the parallelism is limited into each parallel part, but across the parts, the order is deterministic as the parallel part I will allows finish before the parallel part II. Given that conclusion, we can simply separate each parallel part, and consider them as individual parallel parts. As we discussed previously, if we can buffer all the data used in a parallel part, then their all be very limited impacts from the non-determinism, hence, if we have a big

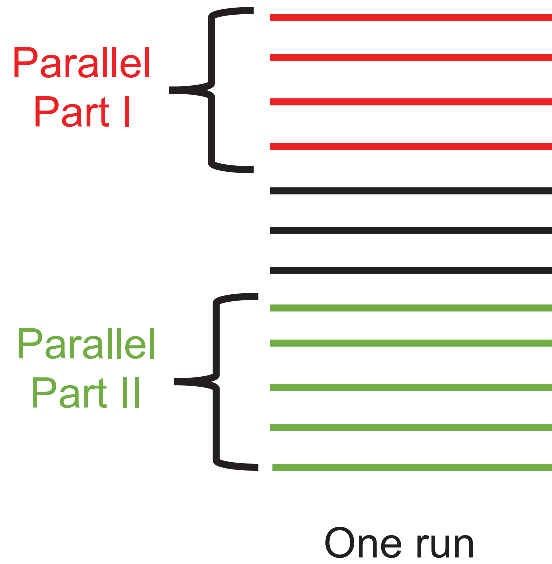


Figure 5.18: An Example of Parallel Parts with Sequential Parts

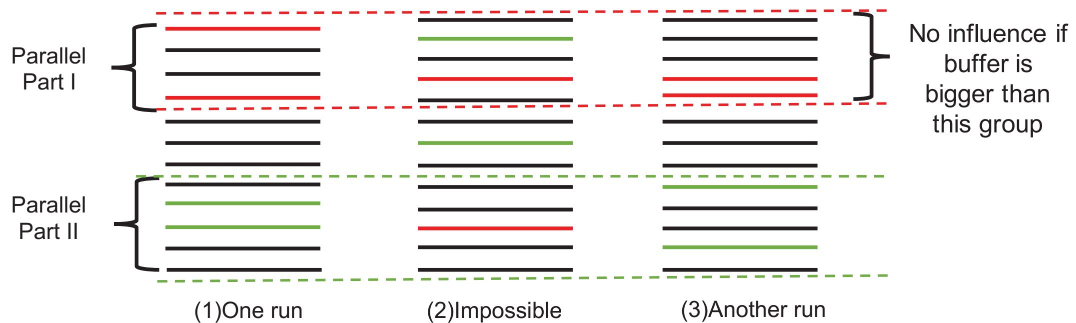


Figure 5.19: The Parallelism Is Not Random Order

enough buffer to buffer all the files used in the largest parallel part, then the non-determinism would not significantly change the overall hit and miss rate. Since the files used in parallel parts can be much smaller compared to the overall file size, the buffer requires to buffer all the files in the largest parallel part can be much smaller.

Parallel Execution is Random Order

We have discussed when we have big enough buffer to buffer all the files used in the largest parallel parts, then there would be no impacts from the non-determinism. In this section, we discuss the situation when we do not even have enough buffer for the largest parallel parts. According to the previous discussion, unlike the case we have big enough buffer, if we can not have enough buffer for all the parallel part, then there can be impacts from the non-deterministic behaviors of parallel applications. After

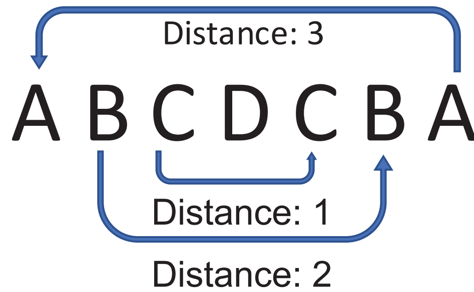


Figure 5.20: An example of reuse distance

the analysis, we found that although we have variations due to the lack of buffer, the impacts are not significant enough to reverse the conclusions from our simulation.

Before the discussion, we first introduce a concept of reuse distance. A reuse distance is the distance between two access to the same data, i.e., reuse. The distance is measured by the number of unique access between two adjacent reuse to the same data. Having a short reuse distance means that data is reused within a small range, i.e., applications have high data locality, and less buffer is required to make sure that data will be buffered. Conversely, long reuse distance means that applications have low data locality, and more buffer is required to make sure that data will be buffered. Figure 5.20 shows an example of reuse distance. We count the reuse distance between the first A and the last A as 3, despite the fact that there are 5 accesses between them, as there are only 3 distinct accesses: B, C, D. Similarly, we count the reuse distance between two Bs as 2, since there are 2 distinct accesses: C and D, and between two Cs there is only one access, D, hence we count it as 1.

In order to understand the impacts from the non-determinism, we first consider how the non-determinism impacts the buffer hit rate, i.e., in what cases would the change of order changes a buffer hit into a buffer miss or vise versa. Figure 5.21 shows an example of two red commands accessing the same data, but the order are changed in two different runs. However, both of them are executed before their data are swapped out or the buffer is full, hence, both of them are buffer hit even though the order have been changed in both cases. Similarly, in Figure 5.22, the orders of red commands are different in the two cases, however, both of them causes buffer miss. From these two cases we notice that the different execution order would not always change a buffer miss to a hit or vise versa, and we find that these two cases are the majorities. If we combine the reuse distance with buffer capacity, we find that when the reuse distance is smaller than the buffer capacity, the second access would be a

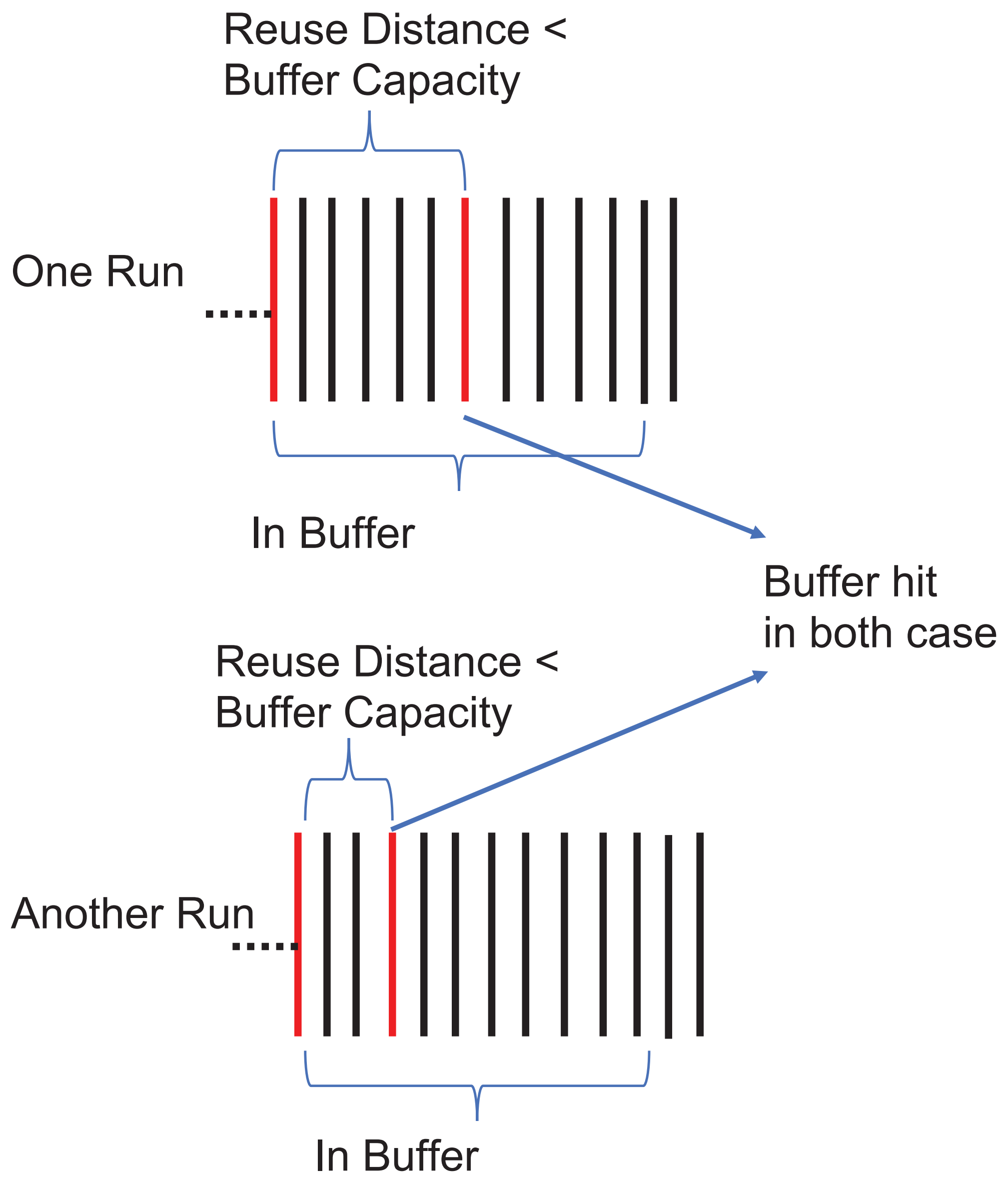


Figure 5.21: Buffer Hit in Both Cases

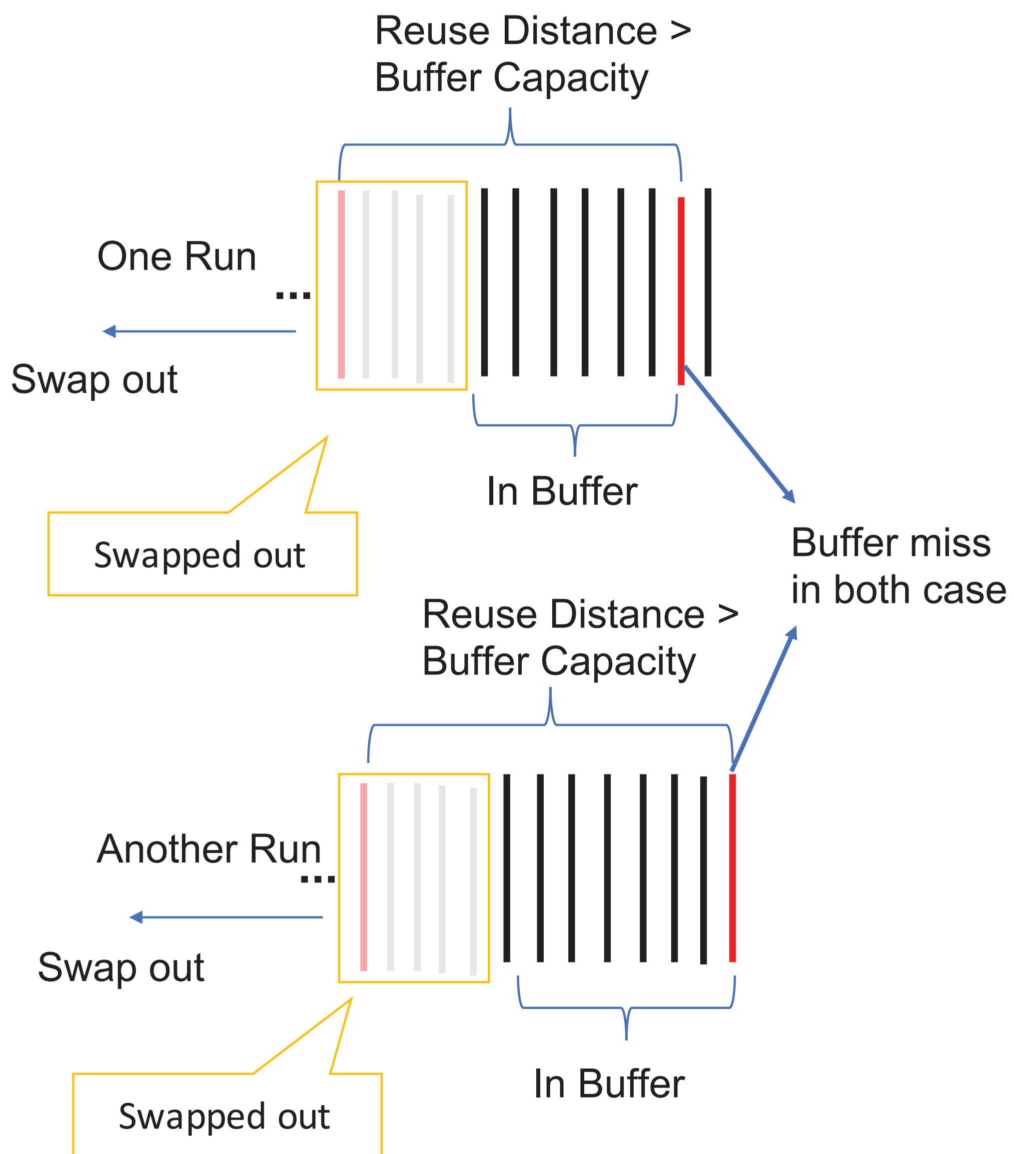


Figure 5.22: Buffer Miss in Both Cases

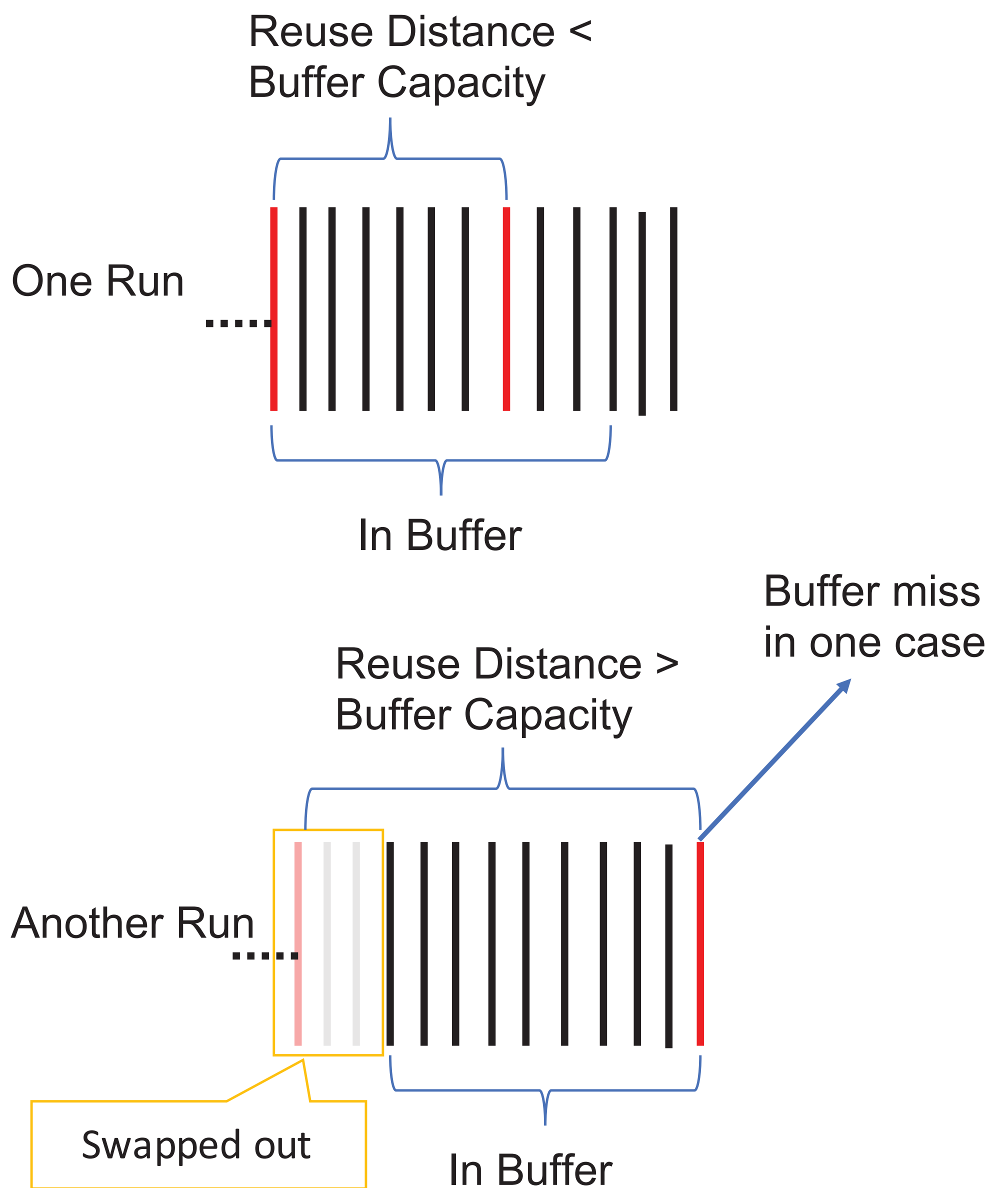


Figure 5.23: Buffer Miss in One Cases

Applications	Buffer Capacity (MB)	Total Access Space (MB)
Montage	4,096	7,500
Similar_pages	32	63
Povray	12	24
Miranda	3,072	7,104
BTIO	5,120	6,800

Table 5.4: HuronFS Configurations for Application

CPU	Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz
Memory	15GB
Network	Ethernet
Filesystem	NFS
Nodes	18

Table 5.5: System Specification

buffer hit, and when the reuse distance is bigger than the buffer capacity, the second access would be a buffer miss. The first example in Figure 5.21 shows the case when the reuse distance is smaller then the buffer capacity, and the second example in Figure 5.22 shows the case when the reuse distance is larger than the buffer capacity.

On the other hand, Figure 5.23 shows another example where the order of two red commands are different in two runs, but in this case one command is executed before the data is swapped out, and the other is executed after, which caused a buffer hit changes to a buffer miss. However, such case can be very rare, because the reuse distance of the two adjacent commands in this case need to be very close to the buffer capacity. Compared to the first two cases, which cover all the reuse distances that are larger or smaller than the buffer capacity, the third case covers very limited reuse distance range that is close to the buffer capacity. With the randomness generated by parallelism, even the accesses have a close reuse distance to the buffer capacity, it would be rare to have all of them fall into the third case. Since the parallelism is random order anyway, we also expect some variations across the executions. Hence, even we have a very limited buffer, as long as the application has a access pattern that can achieve stable performance on such buffer, our simulation are able to simulate the performance.

5.5.3 Empirical Studies

In the previous section, we discussed why and when the parallelism in large scale applications would not affect the results from our simulations significantly.

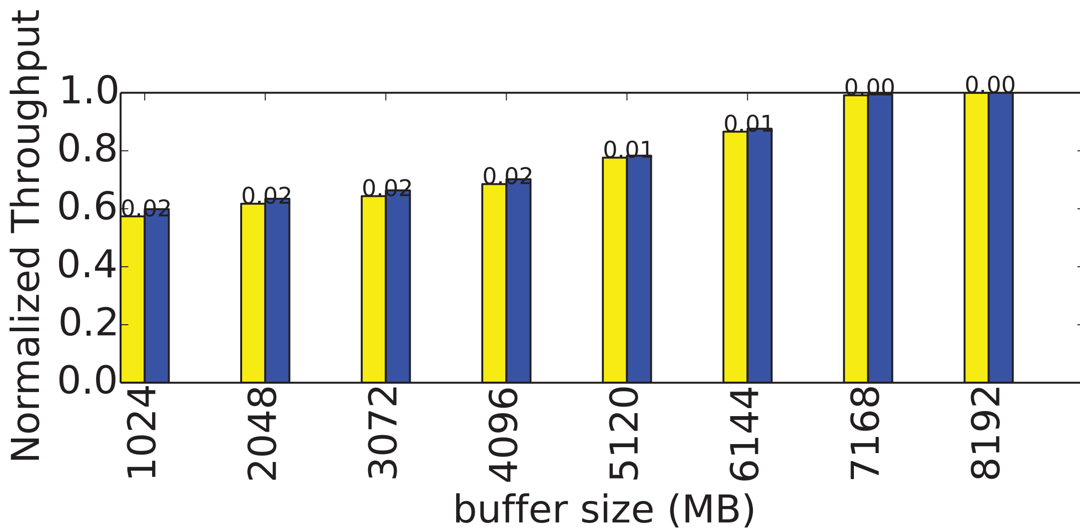


Figure 5.24: Simulation Comparison on Montage

However, we lack of empirical numbers or results on actually how significant the impacts are. In this section, we introduce multiple experiments to give a empirical analysis such impacts. We trace several real world applications with two different configurations : one interacts with background filesystem directly; the other one access the data via HuronFS with limited buffer and no input phrase, which means input data needs to be read from the parallel filesystem to HuronFS before it can be transferred to compute nodes and data swapping happens when buffer is full. In addition to the performance different, we configure HuronFS with no stage-in phrase and with limited buffer capacity to introduce non-determinism via data swapping. Table 5.4 shows the configuration of HuronFS we use for the execution, and Table 5.5 shows the specification of the test bed. Notice that in all the cases, buffer capacities are configured to be smaller than the total access space, which means data swapping will happen in all the execution. By running on two different environments, we obtain two different traces of the same applications. The trace obtained from the direct access give a baseline example of the execution order of the application, while the trace from accessing via HuronFS introduces a different execution order caused by different I/O performance as well as buffer miss and hit. We simulate using both of the two traces and compare the results to see whether the non-determinism would significantly change the simulation results.

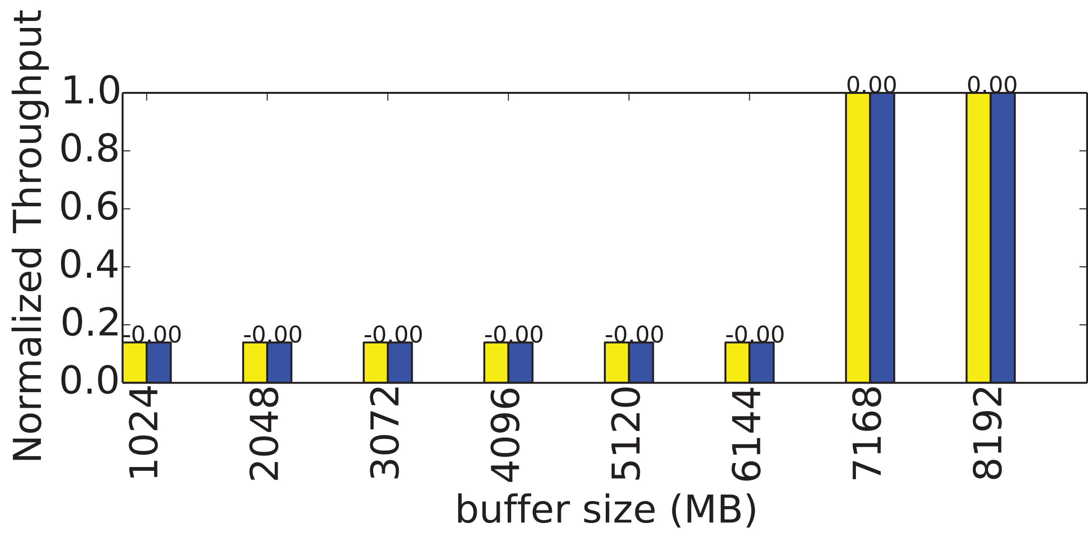


Figure 5.25: Simulation Comparison on BTIO

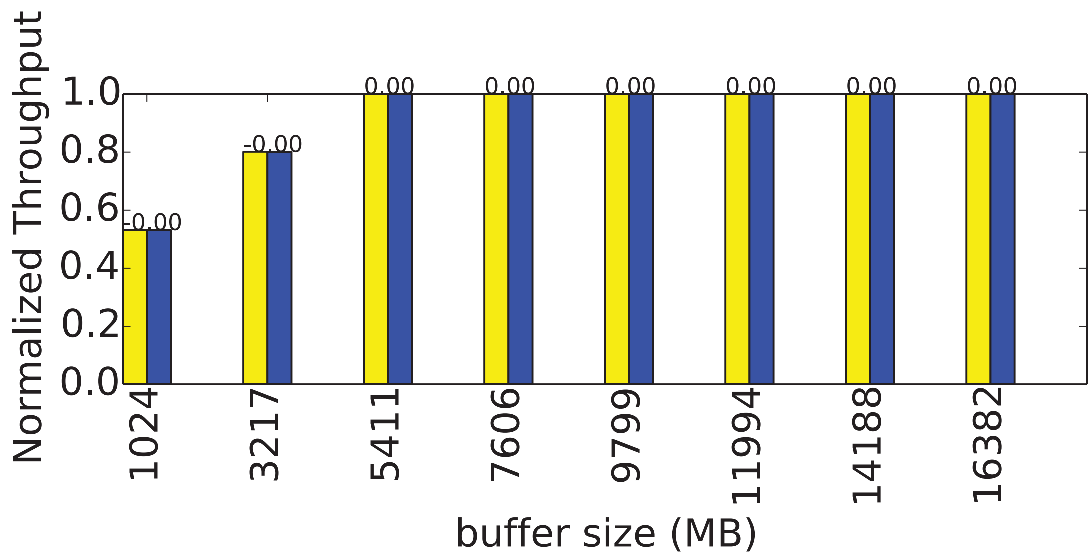


Figure 5.26: Simulation Comparison on Miranda

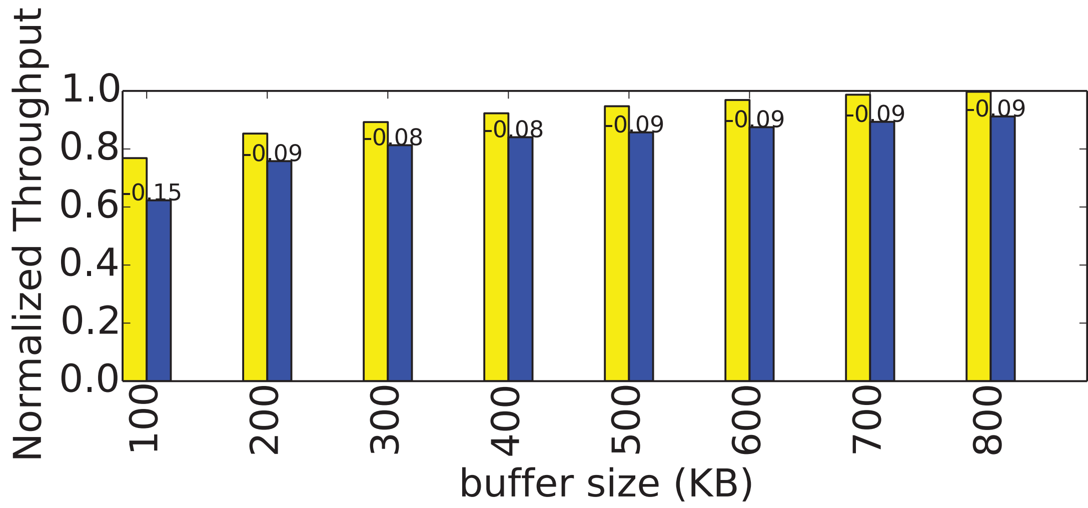


Figure 5.27: Simulation Comparison on Povray

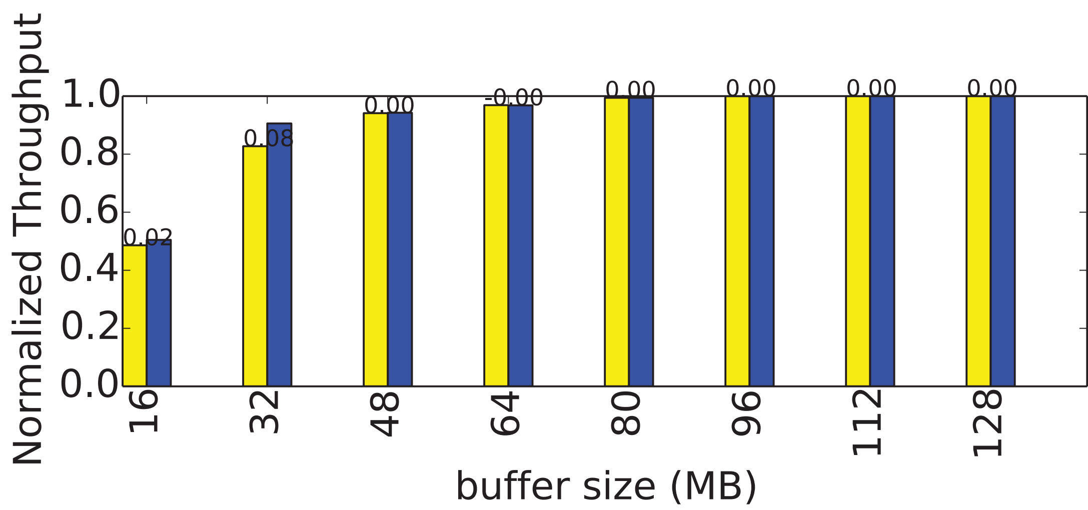


Figure 5.28: Simulation Comparison on Similar Pages

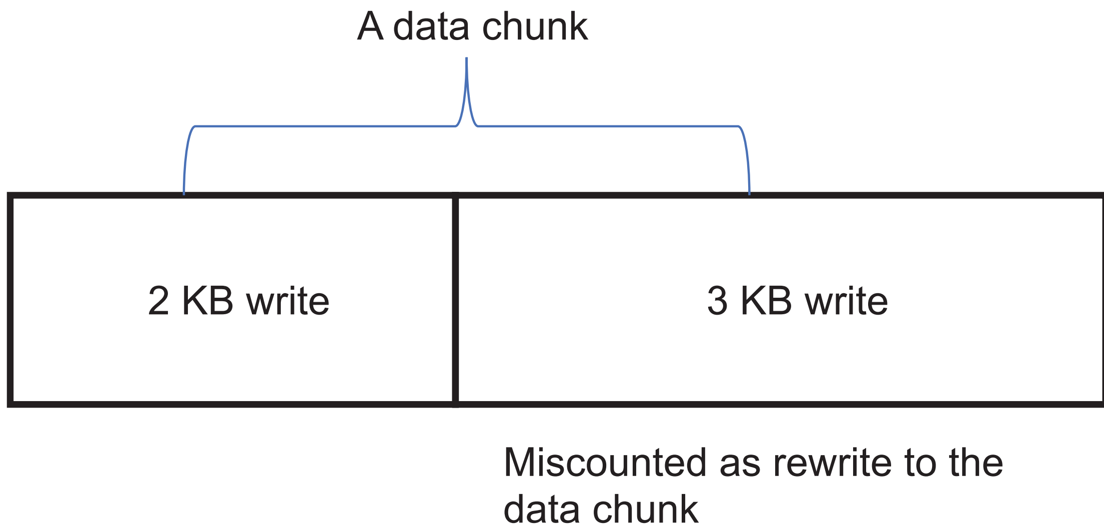


Figure 5.29: An Example of Miscounting

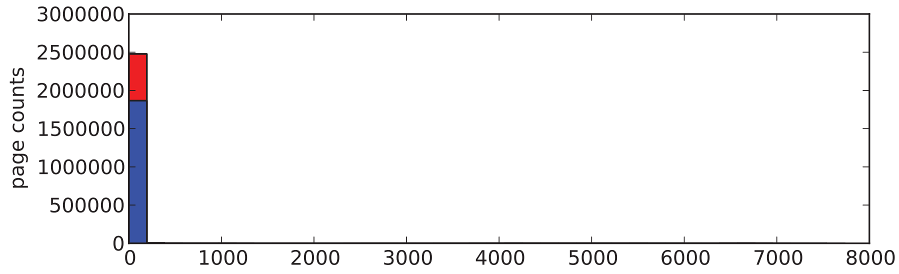
Real Applications

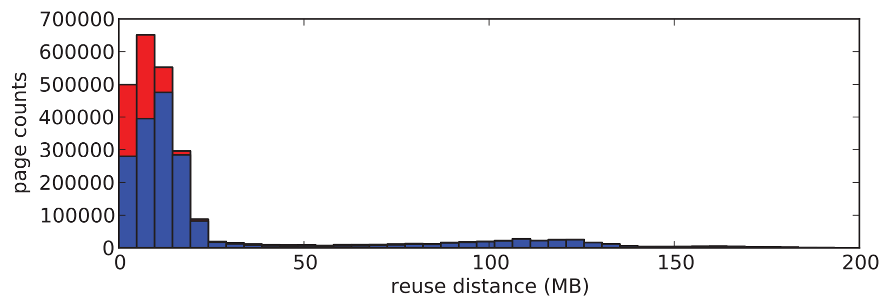
In order to figure out how the actual parallelism in real application can affect the results of simulation is still unclear. Hence, we trace the five applications with the two different configurations. Figure 5.24, 5.25, 5.26, 5.27 and 5.28 show the different on the simulated throughput percentage from our simulations of the two different traces. In the figures, we show the two simulated performance: buffered execution and non-buffered execution in yellow and blue bars, and annotate the difference on the top of the bar.

As we can see from the figures, although the two different runs have difference in execution time and generated two different trace, as we expected, the parallelism has very limited affects on the simulated results. The largest difference is from the Povray at 100 KB, we have 14.54 % difference on the simulated results, while others are around 5%. As we explained in the Section 5.5.2, we expect limited impacts from the parallelism, and the results support our expectation. With the very limited difference from the two traces, we can estimate the expected performance of parallel applications with our simulation.

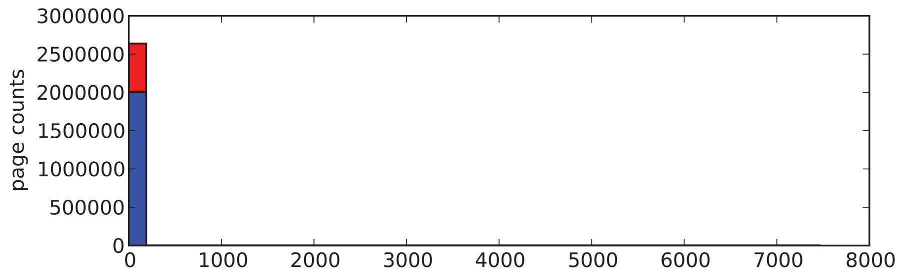
5.5.4 Reuse Distance

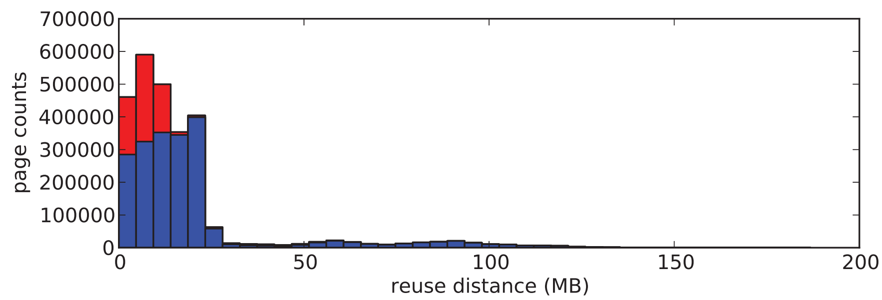
In our hypotheses, we use the reuse distance to explain why different orders can causes insignificant impact on our simulation results. In this section, we calculate the reuse





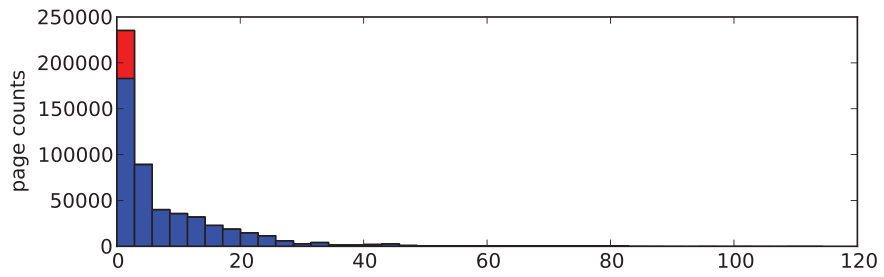
(a) buffered

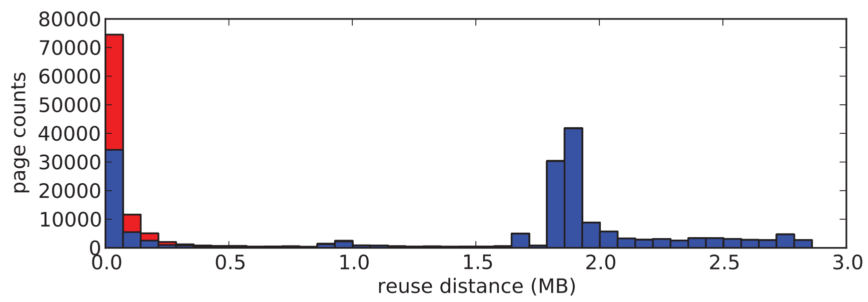




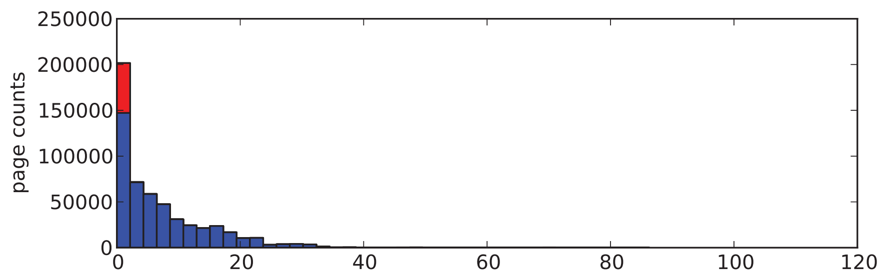
(b) non-buffered

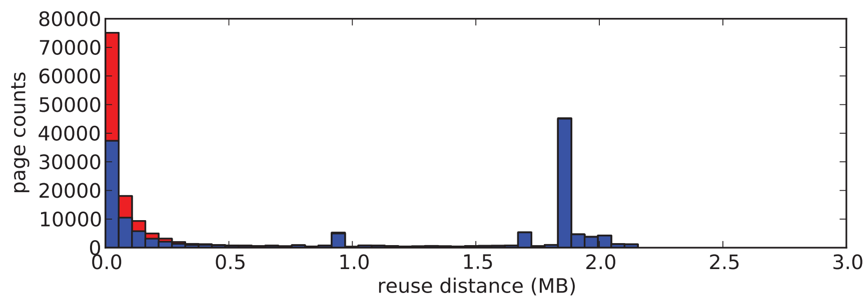
Figure 5.30: Reuse distance of Montage





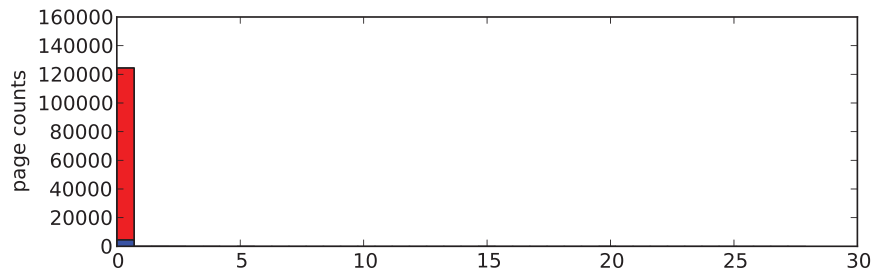
(a) buffered

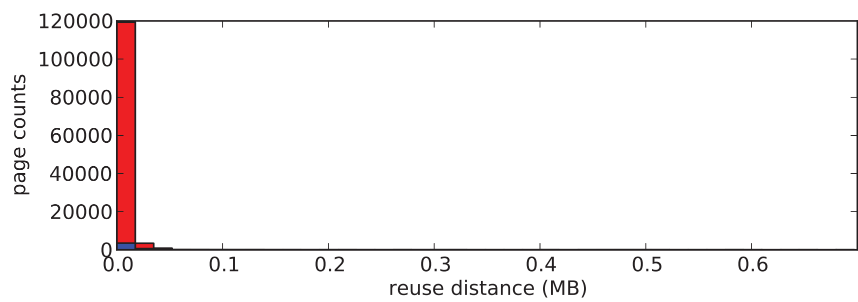




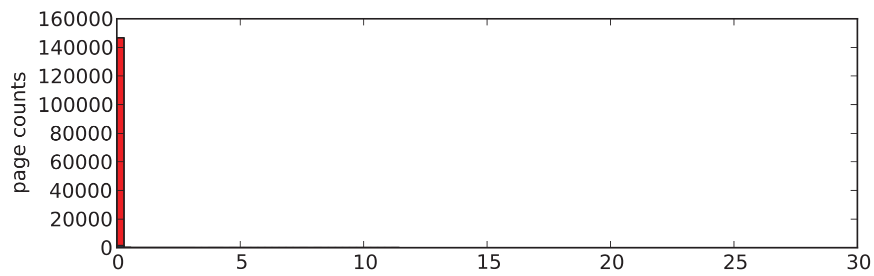
(b) non-buffered

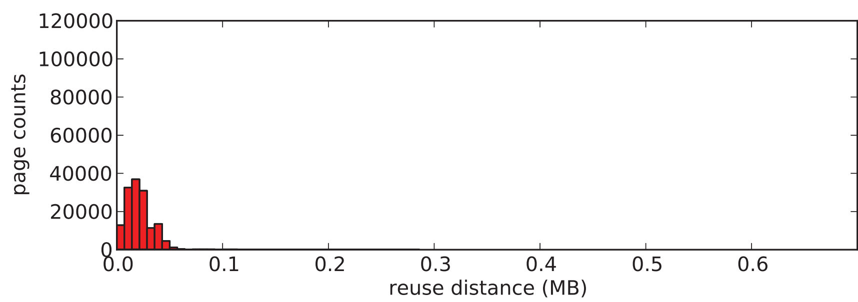
Figure 5.31: Reuse distance of Similar Pages





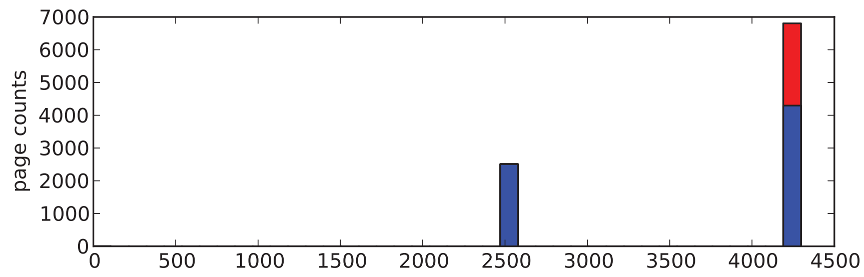
(a) buffered

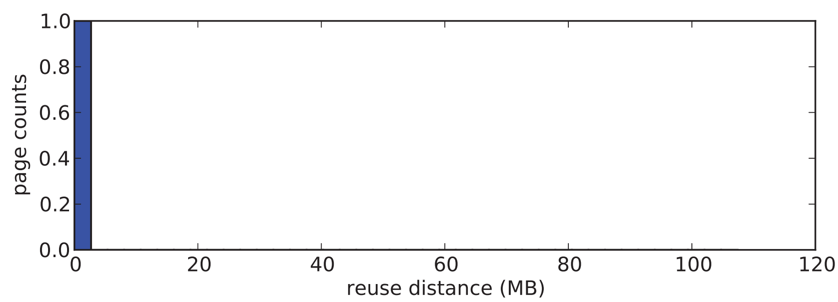




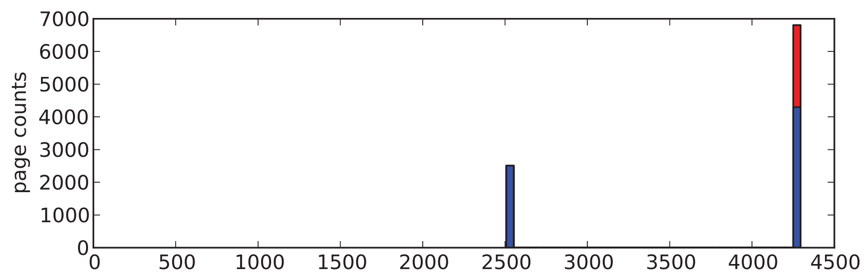
(b) non-buffered

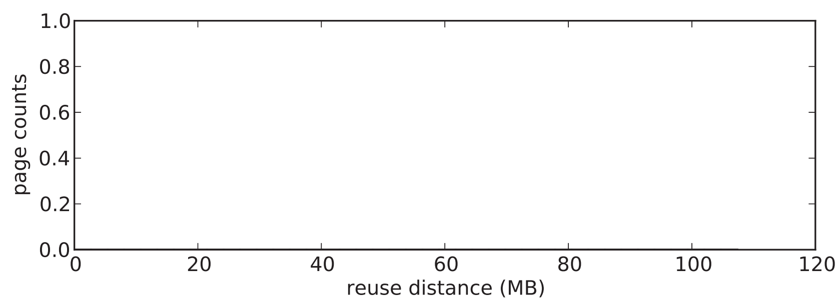
Figure 5.32: Reuse distance of Povray





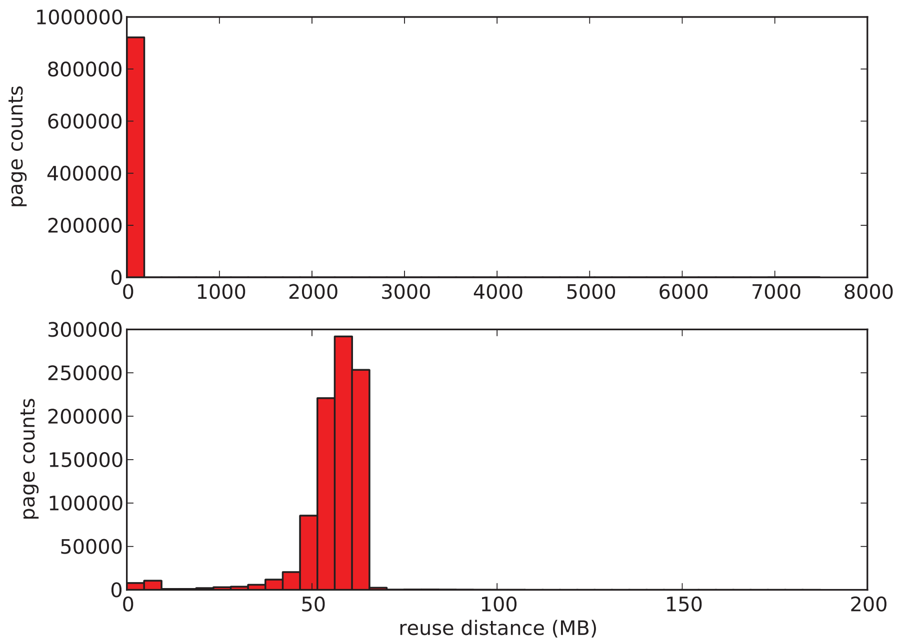
(a) buffered



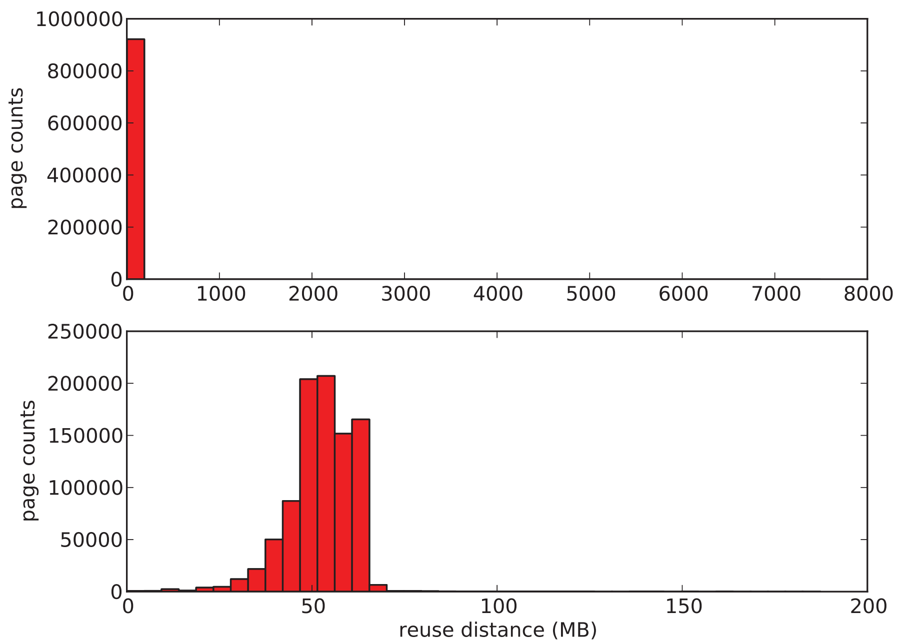


(b) non-buffered

Figure 5.33: Reuse distance of BTIO



(a) buffered



(b) non-buffered

Figure 5.34: Reuse distance of Miranda_IO

distance of all the five applications to show the correctness of our hypotheses. In order to test the correctness of our hypotheses and explain the results we got in the Section 5.5.3 , we calculate reuse distance of both buffered and non-buffered traces, and compare the difference in reuse distance with the performance difference.

When we calculate reuse distance, one problem in is data chunking. Since we split the files into data chunks, we need to count the reuse distance based on the chunks. However, when we count the reuse with such a granularity, due to different data chunking sizes in tracing and our simulations, some appending operations can be mis-recognized as reuses. Figure 5.29 shows an example, during tracing, a sequential read operation of 5 KB from an application can be chunked into a 2 kB and a 3 KB read operations due to some system issues, and generate two different logs for 2 KB and 3 KB respectively. During the simulations, if the split border of the two operations is located within a data chunk, then the second log would be considered as a reuse to the same data chunk by mistake. Such scenario can become even more complicated when we merge different trace files from different nodes, where two split operations can be separated by large amounts of unrelated logs from other nodes. To prevent such miscounting, we keep a record for the last access point of previous access in each file so that when we encounter the successor operation, we can identify whether it is a sequential operation to the previous one, or an operation to different positions.

Figures 5.30˜5.34 show the reuse distance of the five applications respectively. We show the larger scale results of each applications at the top figure. We noticed that for most of the applications, reuse distance are concentrated in the small distance range, so we show a zoomed-in results for the small distance range at the bottom for better details. We separate the read and write in reuse distance calculations, and color read as blue and write as red in the figures.

We can see that all three applications that exhibit high performance under half size buffer: Montage, Povray, and Similar Pages, have very short reuse distances, especially for Povray, which achieves significant improvement with a burst buffer, to have access extremely centralized within the small distance range. Moreover, if we compare the results obtained from the simulator against the reuse distance results, we can see a similar trend.

When we compare the buffered and non-buffer reuse distance, we see significant difference in Montage 5.30, Similar pages 5.31, Povray 5.32, and Miranda 5.34. However, we only see a maximum 14% difference in Povray from Figure 5.27, which has a buffer size that is very close to the main distribution of reuse distance.

The analysis of reuse distance also shows that our hypotheses matches the real cases.

Chapter 6

Related Work

Burst buffer has been used in several research as a new design in storage system to fill the performance gap between node's local storage throughput and parallel file system throughput.

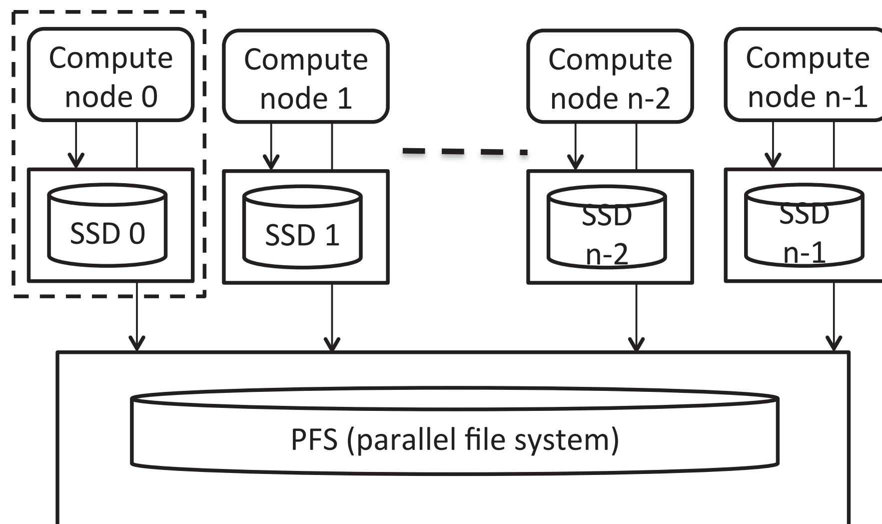


Figure 6.1: Flat buffer C/R system

Besides Liu et al., who proposed burst buffer as a new tier in IBM Blue/Gene storage system, Sato et al. [76] explored effectiveness of burst buffers in checkpoint/restart (C/R) and proposed checkpoint strategy for burst buffers. In traditional C/R strategy, compute nodes write their checkpoint data into parallel file systems, and read checkpoint data to restart after failure occurs. However as the number of compute nodes increases, these compute nodes write into parallel file systems simultaneously, which causes low checkpoint performance due to resource contention. There are some researches using local storage checkpoint strategy to

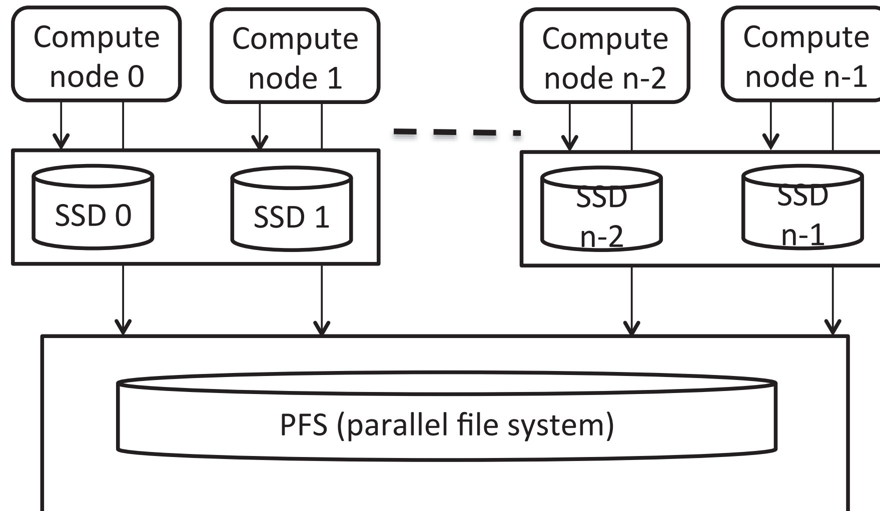


Figure 6.2: Burst buffer C/R system

create scalable C/R system. Figure 6.1 shows such current flat buffer system. In current flat buffer system, each compute node has its dedicated local storage, such as SSD, checkpoint data can store in local storage. Hence such system can scale out as number of nodes increase. However there is a significant disadvantage in such system: checkpoint data lost when node failure occurs, which is the most common failure.

In order to solve these problems, Sato et al. proposed an mSATA SSD based burst buffer system. Figure 6.2 shows architecture of their burst buffer system, in their system, a new burst buffer tier is added into storage hierarchy. They used a small number of additional nodes as burst buffer nodes, and let compute nodes write their checkpoint data into these burst buffers. Since burst buffer nodes are relatively small compared to compute nodes, the probability of lost checkpoint data is decreased and hence it makes the system more reliable. Moreover checkpoint performance can be increased by increasing the number of burst buffer nodes.

The emergence of a cloud computing technology brings a new solution to large scale high performance computing with low initial costs, high accessibility, and flexibility. There are already several researches using public clouds for large scale computation.

Wittek et al. [86] combined an implementation-independent workflow designer with cloud computing to support small institution in ad-hoc peak computing needs. They described a system developed in SHAMAN (Sustaining Heritage Access through Multivalent ArchiviNg) for digital preservation. The system combines a

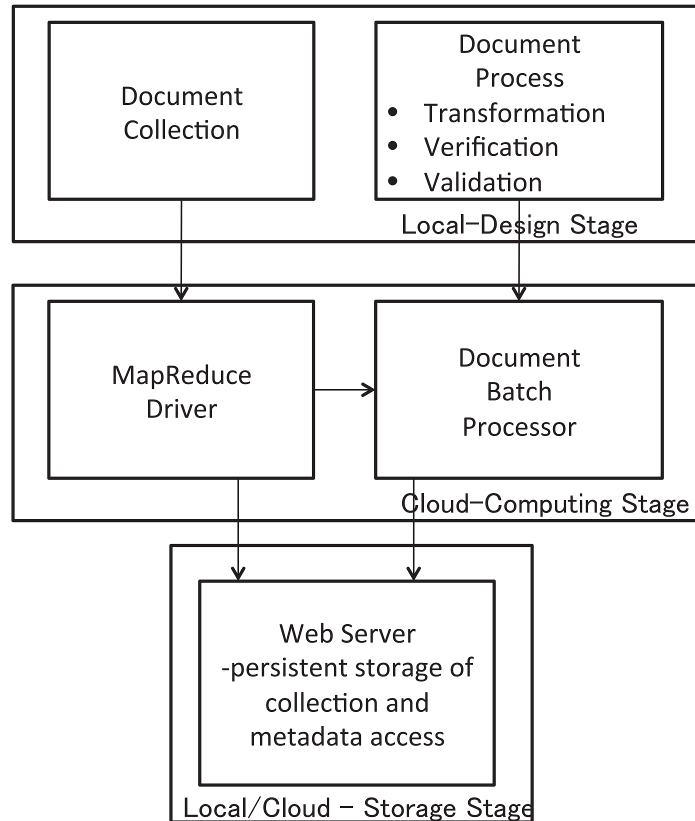


Figure 6.3: Architectural overview of XML processing in the cloud to support digital preservation

XML document process designer with cloud. In their research, they focused on three aspects of digital preservation:

- Migration and Transformation
  This aspect refers to problem of keeping digital data with various data formats more accessible. For example, once the vendor of proprietary file formats stops supporting their softwares, these files become obsolescence. One common solution of such problem is migrating data formats to a more persist format.

- Scalability
  Replacing components or transforming the objects in the historical achieve requires scalability.

- Reusability
  Reusability refers to the verification and exploitation of digital content. Since the new customers of digital content are unable to refer back to the creators, reusability depends on proper descriptions provided by the archive. Document

process preservation helps to create an architecture independent description for the digital content to solve such problem.

These three aspects are not unrelated, migration is a part of reusability. In migration process, even with several optimizations available, process a single large document still takes hours, which prohibits small organizations from these operations due to the computational resources limitation. In order to solve such problems, Wittek et al. considered using cloud and MapReduce technologies for these digital data processing. Figure 6.3 shows the architecture of their XML processing in cloud.

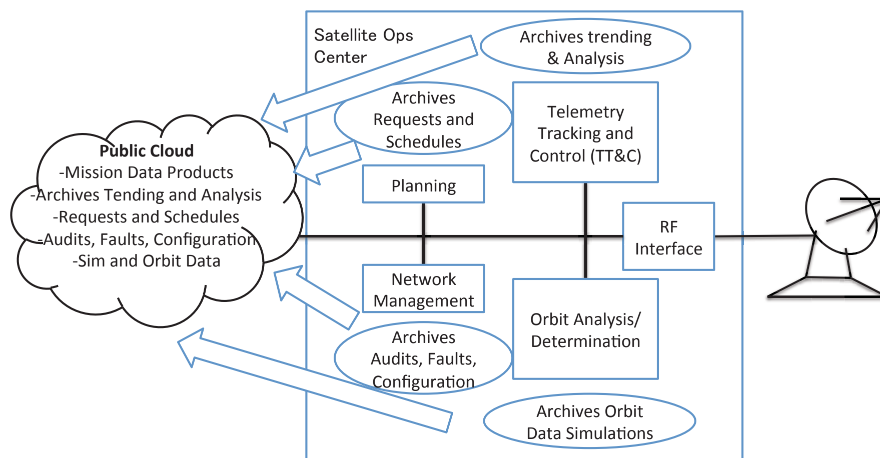


Figure 6.4: Single Satellite Ground System

Anthony et al. [23] discussed the applicability of cloud computing to large-scale satellite ground systems. Most satellite centers need to store the mission data such as historical satellite telemetry, command history information, past data used in orbital calculations, fault/configuration information and system performance data for a long time. Unlike real-time data, this data is not always used on a regular basis and does not have the same retrieval timing requirements. Hence Anthony et al. considered using cloud storage to take advantage of large capacities and cost saving in cloud. Figure 6.4 and 6.5 show two approach using public cloud to store the optional data for single and multiple satellites ground systems.

However, these work only focus on how to run these large scale application on clouds, and do not consider the problem of increased I/O throughput to shared cloud storage.

I/O throughput problem is a critical part in cloud computing and many works have been focusing on this problem. Hovestadt et al. [42] considered another way to improve the I/O performance, they proposed a new adaptive compression scheme for

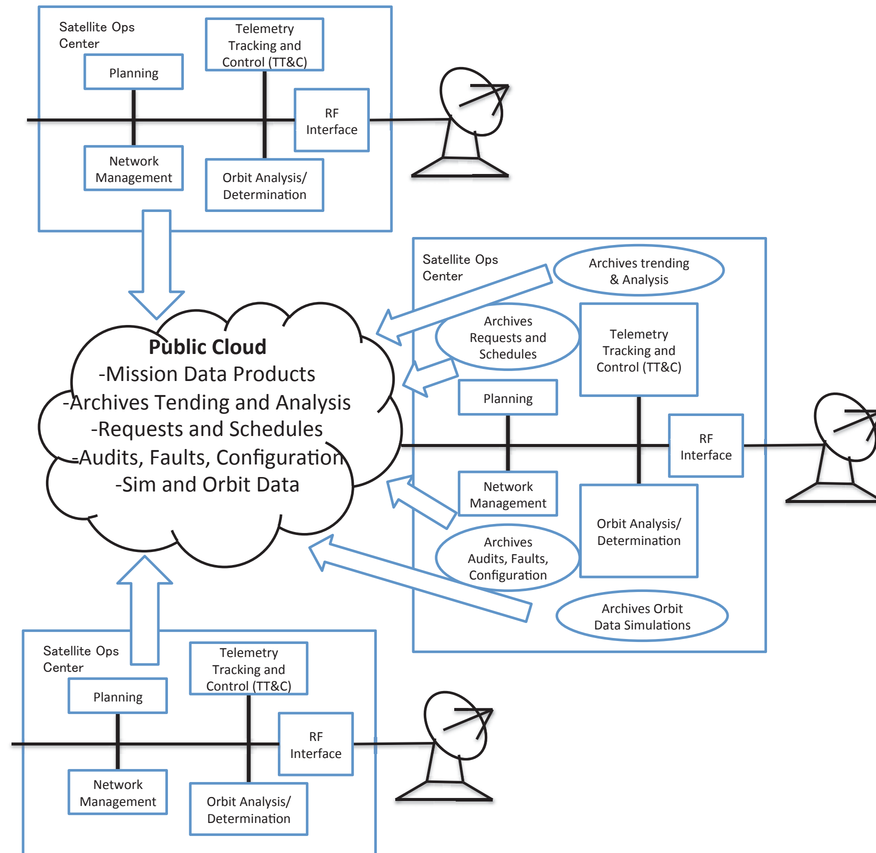


Figure 6.5: Multiple Satellite Ground Systems

virtualized environments and improved the I/O performance by compressing the I/O data. Their idea is to improve the I/O performance by continuously choosing different level compression and applying to the outgoing data stream. Most existing adaptive data compression schemes require calibration and training phase, however on cloud, the information about physical IT infrastructure and co-located virtual machines are not available for users, which leads to poor accuracy. Hence they proposed a new decision model for adaptive data compression. Their decision model has following design goals:

- No training phase:
  Since Most existing adaptive data compression schemes have poor accuracy on cloud due to calibration and training phase, their decision model must not need offline calibration and training phase to be used on cloud.

- No decision based on CPU resources:
  The available CPU resources under high I/O load is easily to be skewed, so their decision model must not rely on that CPU resources.

- Embrace throughput fluctuations:
  Their decision model focus on a granularity level of MB in order to allow for the possible throughput fluctuations.

Algorithm 7 shows the concrete decision algorithm.

Algorithm 7 GetNextCompressionLevel(cdr, pdr, ccl) [42]

$d \leftarrow (cdr - pdr)$
$c \leftarrow c + 1$
$ncl \leftarrow ccl$
if $|d| \leq \alpha \times pdr$ **then**
  No change in application data rate
  if $c \geq 2^{bck[ccl]}$ **then**
    Backoff over, try another compression level
    if $inc = TRUE$ **then**
      $ncl \leftarrow ncl + 1$
    else
      $ncl \leftarrow ncl - 1$
    end if
    $c \leftarrow 0$
  end if
else if $d > 0$ **then**
  Application data rate has improved
  $bck[ccl] \leftarrow bck[ccl]$
  $c \leftarrow 0$
else
  Application data rate has decreased
  $bck[ccl] \leftarrow 0$
  if $inc = TRUE$ **then**
    $ncl \leftarrow ncl - 1$
  else
    $ncl \leftarrow ncl + 1$
  end if
  $c \leftarrow 0$
end if
return ncl

Hongtao Du et al. [34] focused on the performance of meta-data server in cloud environments. They proposed a high throughput system for cloud storage by building meta-data server on the high performance mainframe. Meanwhile, we extend and apply the state-of-the-art burst buffer technology for clouds to accelerate I/O performance. To the best our knowledge, this work is the first

Variable	Meaning
ccl	The compression level applied to the outgoing data stream currently.
ncl	The next compression level shall be applied to the data. The value is based on the algorithm's decision.
c	A counter variable decides how often the decision algorithm has been called since the last change of compression level.
inc	A Boolean variable indicates whether the compression has been increased at its previous change.
bck	An array which stores the backoff values for compression levels.
cdr	The average application data rate. The value has been determined for the last t seconds using the compression level ccl.
pdr	The average application data rate which has been determined for the t seconds before the last t seconds.

Table 6.1: Explanation of the decision alghrithm's variables [42]

explorations of innovating burst buffer technologies in order to solve low I/O throughput problems in clouds.

Some efforts have modelled the performance of burst buffer systems. Paper [77] exhibited an evaluation of the IME burst buffer technology. The authors tested the performance of the burst buffer system and developed a performance model for estimating the performance improvement by using the burst buffer system. The authors also determined the specification of the burst buffer based on the model. However, the target applications used in this paper had only checkpoint-like I/O patterns, where the computational and I/O phase are totally separated. The model was simply based on the amount of data burst buffer can absorb in I/O phase and the amount of data burst buffer can flush to the PFS during the computational phase. On the other hand, our simulations are based on the trace files obtained from large-scale executions of a variety of real applications, and we simulated the behavior of a burst buffer system rather than using a performance model, which is more accurate and precise.

Several simulation work on burst buffer has been published. Peng et. al [45] performed a simulation on the aggregated throughput with/without burst buffer using different number of compute nodes. Paper [28] compared the trade off between NLBB and RSBB on checkpoint/restart by simulation. However, neither of them has simulated complicated I/O in real-world application, nor the impact of different data replacement algorithms.

Data replacement algorithms have been well studied in computer storage systems. Paper [65] proposed a novel data replacement algorithm for Storage Resource Managers (SRMs) that are used in Data Grids, simulated and compared the performance using trace-driven simulation. Paper [69] that exploits the characteristics of flash memory. Paper [41] proposed a new aspect as their algorithm optimizes page miss cost rather than the cache hit rate. Our research on the performance impact from different algorithms guides the design of future burst buffer systems.

Chapter 7

Conclusion and Future Work

With the increasing need of high-performance storage system to support the future large-scale data-intensive applications, we have conducted three different works to improve the performance of the storage systems in two major large-scale cluster, Cloud and HPC centers. We've proposed CloudBB for accelerating I/O performance of data-intensive HPC applications running in clouds. We implemented our proposal using a hybrid of Master-worker and Key-value models, which is capable of simultaneously achieving both high performance and scalability. Our benchmarks have shown that our system accelerates read throughput, write throughput, and metadata operations by up to 10.47, 16.8, and 3.7 times, respectively, on Amazon EC2/S3. Moreover, our system can perfectly solve the eventual consistency issue in Amazon S3, which causes frequent job failures. The executions of Montage and Supernovae show that we can improve performance of a real world data intensive application by up to 4.5 times as well as save 56.3% monetary cost.

We extended our CloudBB into HPC environment in order to help the HPC centers to improve the I/O performance in case of I/O contention and lack of high performance storage. With the help of the high performance framework CCI and multiple-threading, we are able to achieve a comparable performance to the state-of-the-art parallel file systems. By supporting data swapping between the parallel file system, HuronFS is able to handle even a huge data set with limited buffers.

In order to help to fully utilize and optimize the performance of the burst buffer, we studied the performance impacts under different configurations of burst buffer. By simulating the performance of applications, we analyzed three different aspects of configurations: buffer size, swapping granularity and swapping algorithms, According to the simulations results, we found that carefully choosing a good configuration is the key to achieve high performance with burst buffer.

As the future work, we want to further combine our HuronFS with the configuration analysis and automatically configure the HuronFS to match the performance requirements of the applications.

Our studies of design optimization and evaluation of burst buffer in both HPC centers and Cloud helps to alleviate the performance gap between the computation and storage systems, helps to further understand and optimize the burst buffer performance to support the future large-scale data-intensive applications.

Bibliography

[1] Amazon AWS. http://aws.amazon.com/.

[2] Amazon S3 FAQs. http://aws.amazon.com/s3/faqs/?nc2=h_ls.

[3] Amazon simpleDB. http://aws.amazon.com/simpledb/.

[4] Azure. http://azure.microsoft.com/.

[5] Cci. http://cci-forum.com/.

[6] DRBD Users Guide: Replication. https://www.drbd.org/en/doc/users-guide-83/s-three-way-repl.

[7] GXP. http://www.logos.ic.i.u-tokyo.ac.jp/gxp/index.php?FrontPage.

[8] HDFS User Documentation: Data Replication. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Data+Replication.

[9] Himeno benchmark. http://accc.riken.jp/en/supercom/himenobmt/.

[10] HP helion public cloud. http://www.hpcloud.com/.

[11] IBM public cloud. http://www.ibm.com/cloud-computing/us/en/.

[12] Iperf. http://iperf.fr/.

[13] Lustre 2.x documentation. https://build.hpdd.intel.com/job/lustre-manual/lastSuccessfulBuild/artifact/lustre_manual.xhtml#understandinglustre.

[14] Montage. http://montage.ipac.caltech.edu/docs/grid.html.

[15] NPB Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[16] Oracle cloud. https://cloud.oracle.com/home.

[17] Rackspace Cloud Files. http://www.rackspace.com/blog/storage-systems-overview/.

[18] s3fs. https://code.google.com/p/s3fs.

[19] Supernovae. http://www.cluster2008.org/challenge/.

[20] TSUBAME2.5. http://tsubame.gsic.titech.ac.jp/.

[21] Amazon. Amazon API Document. `http://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectPUT.html`.

[22] Amazon. Amazon AWS HPC instance. http://aws.amazon.com/hpc/.

[23] R. Anthony, J. Fritz, and D. Barnhart. Cloud computing applications for large-scale satellite ground systems. In *MILITARY COMMUNICATIONS CONFERENCE, 2011 - MILCOM 2011*, pages 1894–1898, Nov 2011.

[24] G. Bauer, S. Gottlieb, and T. Hoefler. Performance modeling and comparative analysis of the milc lattice qcd application su3_rmd. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 652–659, May 2012.

[25] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. Storage challenges at los alamos national lab. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, April 2012.

[26] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, MW4SOC '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.

[27] R. Bhargava, L. K. John, and F. Matus. Accurately modeling speculative instruction fetching in trace-driven simulation. In *1999 IEEE International Performance, Computing and Communications Conference (Cat. No.99CH36305)*, pages 65–71, Feb 1999.

[28] Lei Cao, Bradley W. Settlemyer, and John Bent. To share or not to share: Comparing burst buffer architectures. In *Proceedings of the 25th High Performance Computing Symposium*, HPC '17, pages 4:1–4:10, San Diego, CA, USA, 2017. Society for Computer Simulation International.

[29] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale i/o workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, Aug 2009.

[30] Tatsuhiro Chiba, Mathijs den Burger, Thilo Kielmann, and Satoshi Matsuoka. Dynamic load-balanced multicast for data-intensive applications on clouds. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 5–14, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Cray. DataWarp. `http://www.cray.com/datawarp`, 2017.

[32] DDN. Datawarp api. https://pubs.cray.com/content/S-2558/CLE%206.0.UP05/xctm-series-datawarptm-user-guide/about-the-datawarp-user-guide.

[33] DDN. Infinite memory engine. https://www.ddn.com/products/ime-flash-native-data-cache/.

[34] Hongtao Du and Zhanhuai Li. Dhfs: A high-throughput heterogeneous file system based on mainframe for cloud storage. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pages 24–28, Dec 2011.

[35] "EBD-CREST". "huronfs (hierarchical, user-level and on-demand filesystem)". https://github.com/EBD-CREST/HuronFS.

[36] FUSE-Team. FUSE. http://fuse.sourceforge.net/, 2015.

[37] Simson L. Garfinkel and Simson L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3, and sqs. Technical report, Center for, 2007.

[38] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[39] Wojciech Golab, Muntasir Raihan Rahman, Alvin Au Young, Kimberly Keeton, Jay J. Wylie, and Indranil Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 28:1–28:2, New York, NY, USA, 2013. ACM.

[40] F. Gropengiesser and K.-U. Sattler. Transactions a la carte - implementation and performance evaluation of transactional support on top of amazon s3. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1082–1091, May 2011.

[41] F. Hou and Y. L. Zhao. A cache management algorithm based on page miss cost. In *2009 International Conference on Information Engineering and Computer Science*, pages 1–4, Dec 2009.

[42] M. Hovestadt, Odej Kao, A. Kliem, and D. Warneke. Evaluating adaptive compression to mitigate the effects of shared i/o in clouds. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1042–1051, May 2011.

[43] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 87–88, June 2012.

[44] S. Jian, L. Zhan-huai, and Z. Xiao. The performance optimization of lustre file system. In *2012 7th International Conference on Computer Science Education (ICCSE)*, pages 214–217, July 2012.

[45] Ioan Raicu Jian Peng, Sughosh Divanji and Michael Lang. "simulating the burst buffer storage architecture on an ibm bluegene/q supercomputer". In *SC16, Supercomputing16*, Nov 2016.

[46] H. Jun, J. Cho, K. Lee, H. Y. Son, K. Kim, H. Jin, and K. Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, May 2017.

[47] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[48] S. Kobbe, L. Bauer, and J. Henkel. Adaptive on-the-fly application performance modeling for many cores. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 730–735, March 2015.

[49] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X. H. Sun. Leveraging burst buffer coordination to prevent i/o interference. In *2016 IEEE 12th International Conference on e-Science (e-Science)*, pages 371–380, Oct 2016.

[50] LANL. Los alamos national laboratory open-source lanl-trace. `http://www.povray.org/`.

[51] LANL. Trinity. `http://www.lanl.gov/projects/trinity/specifications.php`.

[52] N. Liu and C. D. Carothers. Modeling billion-node torus networks using massively parallel discrete-event simulation. In *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1–8, June 2011.

[53] Ning Liu, Christopher Carothers, Jason Cope, Philip Carns, Robert Ross, Adam Crume, and Carlos Maltzahn. Modeling a leadership-scale storage system. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 10–19, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[54] Ning Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In

Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on, pages 1–11, April 2012.

[55] LLNL. The lustre monitoring tool (lmt). https://github.com/llnl/lmt/wiki.

[56] LLNL. mdtest: Mpi-coordinated metadata benchmark test. https://github.com/hpc/ior.

[57] LLNL. Sierra. https://computation.llnl.gov/computers/sierra.

[58] LLNL. Miranda_IO. https://computing.llnl.gov/?set=code&page=sio_downloads, 2016.

[59] P. Lu and K. Shen. Multi-layer event trace analysis for parallel i/o performance tuning. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 12–12, Sept 2007.

[60] I. Mytilinis, D. Tsoumakos, V. Kantere, A. Nanos, and N. Koziris. I/o performance modeling for big data applications over cloud infrastructures. In *2015 IEEE International Conference on Cloud Engineering*, pages 201–206, March 2015.

[61] NASA. Btio. https://www.nas.nasa.gov/publications/npb.html.

[62] NeRSC. Cori. http://www.nersc.gov/users/computational-systems/cori/file-storage-and-i-o/.

[63] S. Nilakantan, S. Lerner, M. Hempstead, and B. Taskin. Can you trust your memory trace? a comparison of memory traces from binary instrumentation and simulation. In *2015 28th International Conference on VLSI Design*, pages 135–140, Jan 2015.

[64] ONL. Summit. https://www.olcf.ornl.gov/for-users/system-user-guides/summit/system-overview/.

[65] Ekow Otoo, Frank Olken, and Arie Shoshani. Disk cache replacement algorithm for storage resource managers in data grids. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–15, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[66] A. Ovsyannikov, M. Romanus, B. V. Straalen, G. H. Weber, and D. Trebotich. Scientific workflows at datawarp-speed: Accelerated data-intensive science using nersc's burst buffer. In *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 1–6, Nov 2016.

[67] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: A viable solution? In *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, DADC '08, pages 55–64, New York, NY, USA, 2008. ACM.

[68] Mayur R. Palankar, Adriana Lamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for Science Grids: A Viable Solution? In *High Performance Distributed Computing, the 2008 international workshop on Data-aware distributed computing*, 2008.

[69] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 234–241, New York, NY, USA, 2006. ACM.

[70] The POV-Team. POV-Ray. http://www.povray.org/, 2017.

[71] rklundt. HP Helion Object Storage Service. http://docs.hpcloud.com/publiccloud/api/object-storage/.

[72] rklundt. IOR HPC benchmark. http://sourceforge.net/projects/ior-sio/.

[73] S. Saini, J. Rappleye, J. Chang, D. Barker, P. Mehrotra, and R. Biswas. I/o performance characterization of lustre and nasa applications on pleiades. In *2012 19th International Conference on High Performance Computing*, pages 1–10, Dec 2012.

[74] K. Sato, K. Mohror, A. Moody, T. Gamblin, B.R. de Supinski, N. Maruyama, and S. Matsuoka. A user-level infiniband-based file system and checkpoint strategy for burst buffers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 21–30, May 2014.

[75] Kento Sato. MUSE. `https://github.com/kento/MUSE`, 2017.

[76] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 19:1–19:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[77] Wolfram Schenck, Salem El Sayed, Maciej Foszczynski, Wilhelm Homberg, and Dirk Pleiter. Evaluation and performance modeling of a burst buffer solution. *SIGOPS Oper. Syst. Rev.*, 50(3):12–26, January 2017.

[78] N. Y. Song, Y. J. Yu, W. Shin, H. Eom, and H. Y. Yeom. Low-latency memory-mapped i/o for data-intensive applications on fast storage devices. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 766–770, Nov 2012.

[79] Titech. Tsubame 3.0 supercomputer. `http://www.t3.gsic.titech.ac.jp/en`.

[80] The university of Tokyo. Reedbush. `https://www.cc.u-tokyo.ac.jp/supercomputer/reedbush/service/Reedbush-EN.pdf`.

[81] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To fuse or not to fuse: Performance of user-space file systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 59–72, Berkeley, CA, USA, 2017. USENIX Association.

[82] Jeffrey Vetter and Chris Chambreau. mpip: Lightweight, scalable mpi profiling, 2004. *http://mpip.sourceforge.net*, 2005.

[83] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable i/o tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 26–31, New York, NY, USA, 2009. ACM.

[84] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. An ephemeral burst-buffer file system for scientific applications. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 807–818, Nov 2016.

[85] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An ephemeral burst-buffer file system for scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 69:1–69:12, Piscataway, NJ, USA, 2016. IEEE Press.

[86] P. Wittek, T. Jacquin, H. Dejean, J.-P. Chanod, and S. Daranyi. Xml processing in the cloud: Large-scale digital preservation in small institutions. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1072–1081, May 2011.

[87] O. Yildiz, A. C. Zhou, and S. Ibrahim. Eley: On the effectiveness of burst buffers for big data processing in hpc systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 87–91, Sept 2017.

[88] Hobin Yoon, A. Gavrilovska, K. Schwan, and J. Donahue. Interactive use of cloud services: Amazon sqs and s3. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 523–530, May 2012.