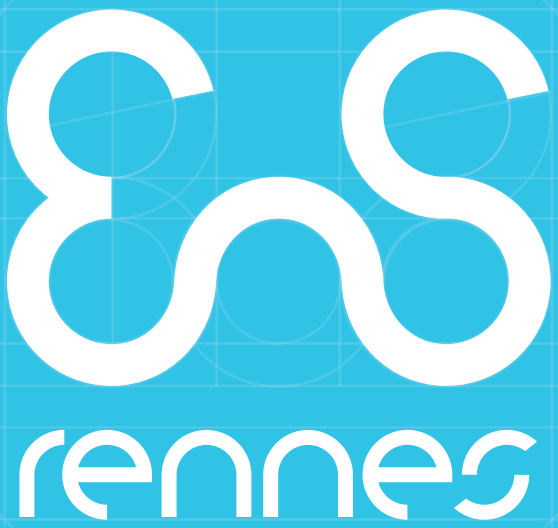
UNIVERSITE BRETAGNE LOIRE



THÈSE / ENS RENNES
Université Bretagne Loire

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE RENNES
Mention : Informatique
École doctorale MathSTIC

présentée par
Orcun YILDIZ
Préparée à l'unité mixte de recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Efficient Big Data Processing on Large-Scale Shared Platforms: Managing I/Os and Failures

Thèse soutenue le 8 décembre 2017
devant le jury composé de :

M. BEAUMONT Olivier / *rapporteur*
Directeur de Recherche, Inria Bordeaux Sud-Ouest, France

M. MONNET Sébastien/ *rapporteur*
Professeur, Polytech' Annecy - Chambéry, France

Mme ARANTES Luciana / *examinatrice*
Maître de Conférences, Université Pierre et Marie Curie, France

M. TAIANI François/ *examinateur*
Professeur, Université de Rennes 1, France

M. ANTONIU Gabriel / *directeur de thèse*
Directeur de Recherche, INRIA Rennes - Bretagne Atlantique, France

M. IBRAHIM Shadi / *co-directeur de thèse*
Chargé de Recherche, INRIA Rennes - Bretagne Atlantique, France

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Contributions . . . 3
1.3 Publications . . . 5
1.4 Organization of the Manuscript . . . 6

2 Background: Big Data Processing on Large-Scale Shared Platforms **7**
2.1 Big Data Processing: An Overview . . . 8
2.1.1 The Deluge of Big Data . . . 8
2.1.2 The MapReduce Programming Model . . . 8
2.1.3 Big Data Processing Frameworks . . . 9
2.1.4 Big Data Processing in Large-Scale Shared Platforms: Early Adoptions 11
2.2 I/O Management in Large-Scale Shared Platforms . . . 14
2.2.1 I/O Latency . . . 14
2.2.2 I/O Interference . . . 15
2.3 Handling Failures at Large Scale . . . 17
2.3.1 Failure-Handling Strategies in Big Data Processing . . . 17
2.3.2 Mitigating Failures in Shared Hadoop Clusters . . . 18
2.4 Discussion: Enabling Efficient Big Data Processing for Large-Scale Shared Platforms . . . 19

3 Characterizing the Performance of Big Data Applications in HPC Systems **21**
3.1 Motivation . . . 22
3.2 Performance Bottlenecks of Big Data Applications on HPC Systems . . . 23
3.2.1 Methodology . . . 23
3.2.2 The Impact of I/O Latency . . . 24
3.2.3 The Impact of Contention . . . 29
3.2.4 The Impact of the File System Configuration . . . 31
3.3 Toward Performing Efficient Big Data Processing on HPC Systems . . . 32
3.4 Related Work . . . 34
3.5 Conclusions . . . 35

4 Zoom on the Root Causes of I/O Interference in HPC Systems **37**
4.1 I/O Interference As a Major Performance Bottleneck . . . 38

4.2 Experimental Insight Into the Root Causes of I/O Interference in HPC Storage Systems . . . 39
4.2.1 Methodology . . . 39
4.2.2 Experimental Results . . . 41
4.3 Unexpected Behaviors: A Flow-Control Issue . . . 51
4.3.1 The Incast Issue . . . 51
4.3.2 From Incast to Unfairness . . . 52
4.3.3 Counter-intuitive Results From a Flow-Control Perspective . . . 54
4.4 Related Work . . . 55
4.5 Conclusions . . . 56

5 Eley: Leveraging Burst Buffers for Efficient Big Data Processing 59
5.1 Limitations of Current Burst Buffer Solutions to Big Data Processing . . . 60
5.2 The Eley Approach . . . 61
5.2.1 Overview of Eley . . . 61
5.2.2 Design Principles . . . 63
5.2.3 Performance Models for HPC and Big Data Applications . . . 63
5.2.4 Interference-Aware Prefetching . . . 65
5.3 Experimental Evaluation . . . 68
5.3.1 Methodology . . . 68
5.3.2 Real System Experiments . . . 71
5.3.3 Simulation Results . . . 73
5.3.4 Summary of the Results . . . 76
5.4 Related Work . . . 77
5.4.1 Early Adoption of Big Data Processing in HPC Systems . . . 77
5.4.2 Extending Burst Buffers for Big Data Applications in HPC Systems . . . 77
5.5 Conclusions . . . 77

6 Chronos: Enabling Fast Failure Recovery in Large Shared Clusters 79
6.1 The Unawareness of Failures in Current Job Schedulers . . . 80
6.2 Fast Failure Recovery with Failure-Aware Scheduling . . . 81
6.2.1 Chronos: Design Principles . . . 81
6.2.2 Work-conserving Preemption . . . 83
6.3 Experimental Evaluation . . . 87
6.3.1 Methodology . . . 88
6.3.2 The Effectiveness of Chronos in Reducing Job Completion Times . . . 89
6.3.3 The Effectiveness of Chronos in Improving Data Locality . . . 90
6.3.4 Impact of Reduce-Heavy Workloads . . . 91
6.3.5 The Effectiveness of the Preemption Technique . . . 91
6.3.6 Overhead of Chronos . . . 92
6.3.7 Chronos and Hadoop Under Multiple Failures . . . 92
6.3.8 Chronos with Aggressive Slot Allocation . . . 92
6.3.9 Discussion of the Results . . . 94
6.4 Related Work . . . 95
6.4.1 Scheduling in Big Data Systems . . . 95
6.4.2 Exploiting Task Preemption in Big Data Systems . . . 95
6.5 Conclusions . . . 96

7 Conclusions and Perspectives **97**
7.1 Achievements 98
7.2 Perspectives 99

8 Appendix **113**

List of Figures

2.1 The MapReduce programming model as originally introduced by Google. . . 9
2.2 Typical storage systems in clouds and HPC systems. 16
2.3 Job statistics of a Hadoop cluster in Carnegie Mellon University. 17

3.1 Performance of Big Data workloads on Spark under data-centric and compute-centric paradigms. 25
3.2 Performance of the Wordcount workload with different input sizes. 26
3.3 CDFs of running times of map tasks in the Wordcount workload. 27
3.4 CDFs of running times of reduce tasks in the Wordcount workload. 28
3.5 Impact of the location of intermediate data on the performance of Sort. 29
3.6 Performance of multiple Wordcount workloads under different paradigms. . 30
3.7 Performance of Wordcount when running alone and together with IOR. . . . 31
3.8 Performance of Wordcount under different configurations of PVFS. 32
3.9 Impact of the memory capacity of the burst buffer on the performance of Wordcount. 34
3.10 Impact of the location of the burst buffer on the performance of Wordcount. . 34

4.1 Typical parallel storage system and potential points of contention. 38
4.2 Influence of the storage device on interference (contiguous pattern). 42
4.3 Influence of the storage device on interference (strided pattern). 43
4.4 Influence of the network interface on interference. 45
4.5 Influence of the network on interference. 45
4.6 Influence of the number of storage servers on interference. 47
4.7 Influence of the targeted storage servers on interference. 48
4.8 Influence of the data distribution policy on interference. 49
4.9 Influence of the request size on interference. 50
4.10 TCP window sizes of I/O requests when the applications is running alone and interfering. 52
4.11 TCP window sizes and progress of the data transfer from one client of each application and one of the servers. 53
4.12 Incast problem with different number of clients. 54

5.1 Naive adoption of burst buffers for Big Data processing in existing studies. . 60
5.2 System overview of Eley. 62

5.3 Use cases of five actions. 67
5.4 Performance comparison between Eley-NoAction and NaiveBB. 71
5.5 Performance comparison between Eley and Eley-NoAction. 72
5.6 Slowdowns observed for the three HPC applications. 74
5.7 Slowdowns observed for the Big Data applications. 75
5.8 Performance of Big Data and HPC applications with Eley approach. 75

6.1 Recovery task waiting times and job completion times of Fifo scheduler under failure. 81
6.2 Chronos overview. 83
6.3 Overview of the three preemption techniques. 84
6.4 Map task preemption. 86
6.5 Reduce task preemption. 87
6.6 Performance comparison for Map-Heavy jobs. 89
6.7 Data locality for Map-Heavy jobs under Chronos, Fifo and Fair Schedulers. . 90
6.8 Job completion times for Reduce-Heavy jobs under Chronos and Fifo scheduler. 91
6.9 Job completion times for Reduce-Heavy jobs under Chronos and Chronos-Kill. 93
6.10 Overhead of Chronos. 93
6.11 Performance comparison of Big Data jobs under double failure. 93
6.12 Job completion times under Fifo, Chronos and Chronos* schedulers with single failure. 94

List of Tables

3.1 Execution time of Sort and its phases under different configurations of PVFS. 32
3.2 Our major findings on the characteristics of Big Data applications on HPC systems. 36

4.1 Time taken by an application running locally to write 2 GB using a contiguous pattern, alone and interfering with another identical application. 41
4.2 Peak interference factor observed by the application for different numbers of storage servers. 47
4.3 Our major findings on the root causes of I/O interference and their implications on performing efficient Big Data processing in HPC systems. 57

5.1 Details of the actions. 66
5.2 Iterations in decision making when choosing the optimal action. 68
5.3 HPC application characteristics for the simulation. 70
5.4 Big Data application characteristics for the simulation. 70
5.5 Applied actions on prefetching during one run of the Sort application. 72
5.6 Average map times of Big Data applications with NaiveBB and Eley-NoAction. 73
5.7 Applied actions on prefetching during the first 5 iterations of the Application 6 and T2 and their respective cost values. 76

Chapter 1

Introduction

Contents

1.1 Context 1
1.2 Contributions 3
1.3 Publications 5
1.4 Organization of the Manuscript 6

1.1 Context

As of 2017, we live in a data-driven world where data-intensive applications are bringing fundamental improvements to our lives in many different areas such as business, science, health care and security. This has boosted the growth of the data volumes (i.e., deluge of Big Data). For instance, International Data Corporation (IDC) Research reports that the amount of data generated in 2016 was 16.1 zettabytes [67]. Same IDC study forecasts that the world will create 163 zettabytes in 2025 which points out the ten-fold increase in the data sizes in less than a decade.

To extract useful information from this huge amount of data, different data processing models have been emerging [30, 69]. Among these models, MapReduce [30] has stood out as the most powerful Big Data processing model, in particular for batch processing. MapReduce, and its open-source implementation Hadoop [53], is adopted in both industry and academia due to its simplicity, transparent fault tolerance and scalability. For instance, Carnegie Mellon University (CMU) is using Hadoop clusters for data analysis on several scientific domains including computational astrophysics, computational neurolinguistics, natural language processing, computational biology and social networking analysis [100]. Besides MapReduce and Hadoop, new data processing frameworks have been introduced [20,

123, 152], such as Spark and Storm with the emphasize on iterative applications, stream processing, machine learning and graph processing. Hereafter, we will call these applications as *Big Data applications*.

Big Data processing is traditionally performed by running these data processing frameworks on clouds. Clouds provide resources at large-scale with a viable cost. Hence, industries and research labs employ large-scale clouds to cope with the gigantic data volumes. For instance, Google is known to employ a cluster with more than 200,000 cores to support its business services and thus improving the user experience [49].

Besides clouds, we have recently witnessed that HPC systems gained a huge interest as a promising platform for performing efficient Big Data processing. HPC systems are equipped with low-latency networks and thousands of nodes with many cores thus have a large potential to run Big Data applications, especially for the ones which require timely responses. For instance, PayPal recently shipped its fraud detection software to HPC systems to be able to detect frauds among millions of transactions in a timely manner [90]. Furthermore, Big Data applications can facilitate performing large-scale simulations on HPC systems (e.g., filtering/analyzing the simulation data). For example, John von Neumann Institute for Computing is trying to couple Big Data applications with scientific simulations in order to tackle societal and scientific grand challenges [86].

Usually, these large-scale platforms (i.e., HPC systems and clouds) are used concurrently by multiple users and multiple applications with the goal of better utilization of resources. For instance, there were up to 61 jobs running simultaneously on Argonne's Intrepid machine [37]. Also, in a Hadoop production cluster with 78 different users at CMU, there were at least 10 jobs running concurrently at 5% of the time [100]. Though benefit of sharing these platforms exist, several challenges are raised when sharing these large-scale platforms, among which I/O and failure management are the major ones for performing efficient data processing on these platforms.

In the context of HPC systems, a high number of Big Data applications would be running concurrently on the same platform and thus sharing the storage system (e.g., parallel file system). Thus, I/O interference will appear as a major problem for the performance of these applications. I/O interference is a well-known problem in HPC systems which often detracts the performance of a single-application from the high performance offered by these systems [37, 89]. Furthermore, Big Data applications will face high latencies when performing I/O due to the necessary data transfers between the parallel file system and computation nodes. Moreover, the impact of both interference and latency will be amplified when HPC systems are to serve as the underlying platform for Big Data applications together with large-scale HPC simulations. Even worse, running Big Data applications on these systems can seriously degrade the performance of these simulations which are considered as first-class citizens on HPC systems. This rises the challenge in being able to perform efficient data processing without impacting the performance of currently running large-scale simulations.

Finally, failures are inevitable in large-scale platforms, especially in clouds. This is because clouds consist of failure-prone commodity machines which makes failures an everyday reality. For instance, Barroso et al. [12] indicated that average mean time to failure of a cluster with 10,000 machines is in the range of few hours. Therefore, fast failure recovery plays a crucial role in performing efficient data processing on the clouds.

1.2 Contributions

This thesis focuses on performance aspects of Big Data processing on large-scale shared platforms. Our objective is to provide I/O management and failure handling solutions that can enable efficient data processing on these platforms.

We first focus on I/O related performance bottlenecks for Big Data applications on HPC systems. We start by characterizing the performance of Big Data applications on these systems. We identify I/O interference and latency as the major performance bottlenecks. Next, we zoom in on I/O interference problem to further understand the root causes of this phenomenon. Then, we propose an I/O management scheme to mitigate the high latencies that Big Data applications can encounter on HPC systems. Moreover, we introduce interference models for Big Data and HPC applications based on the findings we obtain in our experimental study regarding the root causes of I/O interference. Finally, we leverage these models to minimize the impact of interference on the performance of Big Data and HPC applications.

Second, we focus on the impact of failures on the performance of Big Data applications by studying failure handling in shared MapReduce clusters. We introduce a failure-aware scheduler which enables fast failure recovery thus improving the application performance. These contributions can be summarized as follows.

Characterizing the Performance of Big Data Applications in HPC Systems

There is a recent trend towards performing Big Data processing on HPC systems. Although these systems offer a rich set of opportunities for Big Data processing (e.g., high-speed networks, high computing power and large memories), they are traditionally designed for compute-intensive applications rather than data-intensive ones. Hence, it is not trivial to perform efficient data processing on HPC without understanding the performance characteristics of Big Data applications on these systems. To this end, we conduct an experimental campaign to provide a clearer understanding of the performance of Spark, the *de facto* in-memory data processing framework, on HPC systems. We run Spark using representative Big Data workloads on Grid'5000 testbed to evaluate how the latency, contention and file system's configuration can influence the application performance. Our experimental results reveal that I/O interference and latency are the major performance bottlenecks when running Big Data applications on HPC systems. We also discuss the implications of these findings and draw attention to I/O management solutions (e.g., burst buffers) to improve the performance of Big Data applications on HPC systems.

Investigating the Root Causes of I/O Interference in HPC Systems

As HPC systems are shared by high number of concurrent applications, I/O interference appears as a major performance bottleneck for Big Data applications. I/O interference is a well known problem in HPC which is defined as the performance degradation observed by any single application performing I/O in contention with other applications running on the same platform. This interference problem is becoming more important every day with the growing number of concurrent applications (HPC and Big Data applications) that share these platforms. Our ultimate goal is to perform interference-aware Big Data processing on

these platforms. In this direction, understanding the root causes of I/O interference problem becomes crucial. To this end, we conduct an extensive experimental campaign by using microbenchmarks on the Grid'5000 testbed to evaluate how the applications' access pattern, the network components, the file system's configuration, and the backend storage devices influence I/O interference. Our studies reveal that in many situations interference is a result of the bad flow control in the I/O path, rather than being caused by some single bottleneck in one of its components. We further show that interference-free behavior is not necessarily a sign of optimal performance. To the best of our knowledge, our work provides the first deep insight into the role of each of the potential root causes of interference and their interplay. This work, partially carried out during a 3-month internship at Argonne National Labs, led to a publication at the IPDPS'16 conference (see [144]).

Leveraging Burst Buffers in HPC Systems to Enable Efficient Big Data Processing

Developing smart I/O management solutions will be of utmost importance for performing efficient data processing on HPC systems. Burst Buffers (BBs) are an effective solution for reducing the data transfer time and the I/O interference in HPC systems. Extending Burst Buffers to handle Big Data applications is challenging because BBs must account for the large data sizes and the QoS (i.e., performance requirements defined by the user) of HPC applications – which are considered as first-class citizens in HPC systems. Existing BBs focus on only intermediate data and incur a high performance degradation of both Big Data and HPC applications. We propose *Eley*, a burst buffer solution that helps to accelerate the performance of Big Data applications while guaranteeing the QoS of HPC applications. To achieve this goal, Eley embraces interference-aware prefetching technique that makes reading the input data faster while controlling the interference imposed by Big Data applications on HPC applications. We evaluate Eley with both real system experiments on Grid'5000 testbed and simulations using a wide range of Big Data and HPC applications. Our results demonstrate the effectiveness of Eley in obtaining shorter execution time of Big Data applications while maintaining the QoS of HPC applications. Part of this work was published at the CLUSTER'17 conference (see [147]).

Enabling Fast Failure Recovery in Large Shared Clusters

Failures are quite common for large-scale platforms, in particular for clouds which consist of commodity machines. Thus, failure handling plays a crucial role in performing efficient Big Data processing. In this direction, we aim to improve the application performance under failures in shared Hadoop (popular open-source implementation of MapReduce) clusters. Currently, Hadoop handles machine failures by re-executing all the tasks of the failed machines (i.e., by executing recovery tasks). Unfortunately, this elegant solution is entirely entrusted to the core of Hadoop and hidden from Hadoop schedulers. The unawareness of failures therefore may prevent Hadoop schedulers from operating correctly towards meeting their objectives (e.g., fairness, job priority) and can significantly impact the performance of Big Data applications. We addressed this problem by introducing *Chronos*, a failure-aware scheduling strategy that enables an early yet smart action for fast failure recovery while still operating within a specific scheduler objective. Upon failure detection, rather than waiting an uncertain amount of time to get resources for recovery tasks, Chronos leverages a

lightweight preemption technique to carefully allocate these resources. In addition, Chronos considers data locality when scheduling recovery tasks to further improve the performance. We demonstrate the utility of Chronos by combining it with Fifo and Fair schedulers. The experimental results show that Chronos recovers to a correct scheduling behavior within a couple of seconds only and reduces the job completion times by up to 55% compared to state-of-the-art schedulers. This work led to publications at the BigData'15 conference (see [146]) and FGCS journal (see [145]).

1.3 Publications

Journals

- **Orcun Yildiz**, Shadi Ibrahim, Gabriel Antoniu. *Enabling Fast Failure Recovery in Shared Hadoop Clusters: Towards Failure-Aware Scheduling*, Journal of the Future Generation Computer Systems **(FGCS)**, 2016.
- Matthieu Dorier, **Orcun Yildiz**, Shadi Ibrahim, Anne-Cécile Orgerie, Gabriel Antoniu. *On the Energy Footprint of I/O Management in Exascale HPC Systems*, Journal of the Future Generation Computer Systems **(FGCS)**, 2016.
- Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, **Orcun Yildiz**, Shadi Ibrahim, Tom Peterka, Leigh Orf. *Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations*, ACM Transactions on Parallel Computing **(TOPC)**, 2016.

International Conferences

- **Orcun Yildiz**, Amelie Chi Zhou, Shadi Ibrahim. *Eley: On the Effectiveness of Burst Buffers for Big Data Processing in HPC systems*, in Proceedings of the 2017 IEEE International Conference on Cluster Computing **(CLUSTER '17)**, *Hawaii, September 2017.* CORE Rank A (acceptance rate 31%).
- **Orcun Yildiz**, Matthieu Dorier, Shadi Ibrahim, Rob Ross, Gabriel Antoniu. *On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems*, in Proceedings of the 2016 IEEE International Parallel & Distributed Processing Symposium **(IPDPS '16)**, *Chicago, May 2016.* CORE Rank A (acceptance rate 23%).
- **Orcun Yildiz**, Shadi Ibrahim, Tran Anh Phuong, Gabriel Antoniu. *Chronos: Failure-Aware Scheduling in Shared Hadoop Clusters*, in Proceedings of the 2015 IEEE International Conference on Big Data **(BigData '15)**, *Santa Clara, October 2015.* (acceptance rate 35%).

Workshops at International Conferences

- **Orcun Yildiz**, Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu. *A Performance and Energy Analysis of I/O Management Approaches for Exascale Systems*, in Proceedings of the

2014 Data-Intensive Distributed Computing **(DIDC '14)** workshop, held in conjunction with the 23rd International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC '14), *Vancouver, June 2014.*

1.4 Organization of the Manuscript

The rest of this manuscript is organized in six chapters.

Chapter 2 presents the context of this thesis. It introduces the Big Data processing paradigm and presents its applications and platforms. Next, it lists the challenges associated with Big Data processing at large-scale shared platforms and discusses the potential ways to mitigate these challenges that drove our contributions towards performing efficient data processing on these platforms.

Our work aims to provide I/O management and failure handling solutions in order to perform efficient data processing at large-scale shared platforms. Chapters 3 to 5 focus on contributions related to the I/O management in HPC systems. Chapter 3 starts by characterizing the performance of Big Data applications on these systems. It shows that high I/O latency and I/O interference are the major performance bottlenecks in HPC systems. Then, Chapter 4 dives into the I/O interference phenomenon in HPC systems in order to explore its root causes. Based on our observations from these chapters, we introduce the Eley approach in Chapter 5 in order to alleviate the impact of the aforementioned problems.

Chapter 6 is dedicated to the failure handling in clouds. It studies the effect of failures on the performance of Big Data applications and introduces our Chronos approach. Chronos is a failure-aware scheduler that aims to provide a fast recovery in case of failures.

Finally, Chapter 7 concludes this thesis and summarizes our contributions and the perspectives brought by these contributions.

Chapter 2

Background: Big Data Processing on Large-Scale Shared Platforms

Contents

2.1 Big Data Processing: An Overview 8
2.1.1 The Deluge of Big Data 8
2.1.2 The MapReduce Programming Model 8
2.1.3 Big Data Processing Frameworks 9
2.1.4 Big Data Processing in Large-Scale Shared Platforms: Early Adoptions 11
2.2 I/O Management in Large-Scale Shared Platforms 14
2.2.1 I/O Latency 14
2.2.2 I/O Interference 15
2.3 Handling Failures at Large Scale 17
2.3.1 Failure-Handling Strategies in Big Data Processing 17
2.3.2 Mitigating Failures in Shared Hadoop Clusters 18
2.4 Discussion: Enabling Efficient Big Data Processing for Large-Scale Shared Platforms 19

THIS chapter aims to draw a picture of Big Data processing on large-scale shared platforms. First, it introduces MapReduce as the state-of-the-art Big Data processing paradigm and lists the most commonly adopted data processing frameworks. Then, it details the I/O management and failure handling approaches in Big Data processing and the respective challenges. Finally, we discuss the potential ways to mitigate these challenges in order to perform efficient Big Data processing on large-scale shared platforms.

2.1 Big Data Processing: An Overview

2.1.1 The Deluge of Big Data

Data is a driving power in almost every aspect of our lives and thus large amounts of data generated everyday. For instance, International Data Research report [67] estimates that the global data volume subject to data analysis will grow by a factor of 50 to reach 5.2 zettabytes in 2025. This huge growth in the data volumes, the deluge of Big Data, results in a big challenge in managing, processing and analyzing these gigantic data volumes.

Some of these data is generated at a high velocity from various sources such as Internet of Things, network sensors and social networks. As an example from social networks, more than 44 million messages sent and 486,000 photos shared on Whatsapp per minute [48]. These large data volumes with a high velocity mostly require fast processing. Some use cases from our everyday lives which require fast processing include fraud detection, national security, stock market management and customer service optimization. For instance, PayPal managed to save 710 million dollars in their first year thanks to the fast data processing for fraud detection [90].

To benefit from this huge amount of data, while addressing the aforementioned challenges (i.e., Big Data challenges), different data processing models have been emerged [30, 69]. Due to its simplicity, transparent fault tolerance and scalability, MapReduce [30] has become the most powerful Big Data processing model, in particular for batch processing. For instance, Yahoo! claimed to have the world's largest MapReduce cluster [54] with more than 100000 CPUs in over 40000 machines running MapReduce jobs.

Recently, several new data processing frameworks have been introduced such as HaLoop [20], Storm [123] and Spark [152] which extend the MapReduce programming model. These frameworks focus on different type of applications – beyond batch processing – such as stream data processing, iterative applications, graph processing and query processing. For example, Netflix has a Spark cluster of over 8000 machines processing multiple petabytes of data in order to improve the customer experience by providing better recommendations for their streaming services [122]. Hereafter, we will detail the MapReduce paradigm and present some of the most commonly adopted Big Data processing frameworks.

2.1.2 The MapReduce Programming Model

MapReduce is a data processing model for solving many large-scale computing problems [30]. The MapReduce abstraction is inspired by the map and reduce functions, which are commonly used in functional languages. The MapReduce programming model allows users to easily express their computation as map and reduce functions. The map function, written by the user, processes a key/value pair to generate a set of intermediate key/value pairs. The reduce function, also written by the user, merges all intermediate values associated with the same intermediate key to generate the final output. Parallelism of map and reduce functions gives users the opportunity to have fast and scalable data processing. Besides parallelism, data locality and fault tolerance are the other key aspects of MapReduce abstraction for achieving efficient Big Data processing. Next, we describe some of the widely

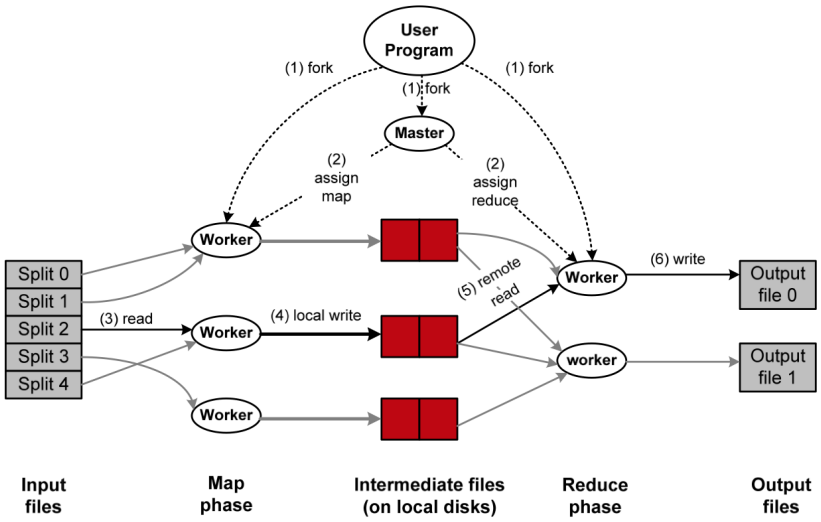


Figure 2.1: The MapReduce programming model as originally introduced by Google [62].

adopted Big Data processing frameworks and present how these frameworks applies these key aspects in their implementation.

2.1.3 Big Data Processing Frameworks

Hadoop

Hadoop [53], the popular open-source implementation of MapReduce, is used to process massive amounts of data. Hadoop is developed primarily by Yahoo!, where it processes hundreds of terabytes of data on at least *10,000 cores*, and is now used by other companies, including Facebook, Amazon, Last.fm, and the New York Times [54].

Hadoop implements the MapReduce programming model which is illustrated in Figure 2.1. Users submit jobs consisting of map and reduce functions in order to perform data processing. These jobs are further divided into tasks which is the unit of computation in Hadoop. Inputs and outputs of these jobs are stored in a distributed file system (i.e., Hadoop Distributed File System (HDFS) [53]). In the map phase, map tasks read the input blocks and generate the intermediate results by applying the user defined map function. These intermediate results are stored on the compute node where the map task is executed. In the reduce phase, each reduce task fetches these intermediate results for the key-set assigned to it and produces the final output by aggregating the values which have the same key. Next, we describe this job execution in detail.

Job execution in Hadoop. In Hadoop, job execution is performed with a master-slave configuration. JobTracker, Hadoop master node, schedules the tasks to the slave nodes and monitors the progress of the job execution. TaskTrackers, slave nodes, run the user defined map and reduce functions upon the task assignment by the JobTracker. Each TaskTracker has a certain number of map and reduce slots which determines the maximum number of map and reduce tasks that it can run. Communication between master and slave nodes is

done through heartbeat messages. At every heartbeat, TaskTrackers send their status to the JobTracker. Then, JobTracker will assign map/reduce tasks depending on the capacity of the TaskTracker and also by considering the data locality (i.e., executing tasks on the machines where the input data resides) which is a key aspect in MapReduce abstraction. To do so, JobTracker will assign map tasks to the TaskTrackers with the input data on it, among the ones with empty slots. By sustaining high data locality, Hadoop can mitigate the I/O latency and thus improve the application performance. Next, we discuss Hadoop's fault tolerance mechanism which is another important factor for the application performance.

Fault tolerance in Hadoop. When the master is unable to receive heartbeat messages from a node for a certain amount of time (i.e., failure detection timeout), it will declare this node as failed [32, 60]. Then, currently running tasks on this node will be reported as failed. Moreover, completed map tasks will also be reported as failed since these outputs were stored on that failed node, not in the distributed file system as reducer outputs. To recover from the failures, Hadoop will try to re-execute all the tasks of failed nodes (i.e., executing recovery tasks) on any healthy node as soon as possible. When there is an empty slot on a healthy node, Hadoop first will launch the cleanup task for the recovery task. Cleanup tasks try to ensure that failure will not affect the correct execution of the MapReduce job by deleting the outputs of failed tasks. When the cleanup task is completed, recovery task can run on the healthy node with an empty slot.

Spark

Hadoop has become successful for a large class of Big Data applications but it provides a limited support for some set of applications including iterative applications and interactive analytics [20, 23, 152]. Spark therefore has been emerging as an efficient Big Data processing framework that supports a wide range of applications successfully [152].

Spark introduces a new abstraction, Resilient Distributed Datasets (RDDs) [151], which represent the input data partitions that is distributed across the cluster. RDDs provide an interface that allows users to apply coarse-grained transformations on these input partitions. These transformations include MapReduce-like operations (e.g., map, reduce, collect). As Hadoop, Spark relies on a distributed storage system (e.g., HDFS) to store the input and output data of the jobs submitted by users. However, unlike Hadoop, Spark allows RDDs to be cached in the memory and therefore intermediate data between different iterations of a job can be reused efficiently. This reduces the number of costly disk I/O accesses to the distributed storage system. This memory-resident feature of Spark is particularly essential for some Big Data applications such as iterative machine learning algorithms which intensively reuse the results across multiple iterations of a MapReduce job. Next, we describe the job execution in Spark.

Job execution in Spark. Similarly to Hadoop, job execution is performed in a master-slave configuration. Users express their jobs as driver programs which are connected to a cluster of slaves. These programs are expressed in terms of RDDs on which users can invoke actions such as reduce, collect and count. The driver, Spark master node, assigns tasks to slaves using delay scheduling [150] in order to mitigate the I/O latency by sustaining high data

locality. To this end, when the delay scheduler is not able to launch a local task, it waits for some time until it finds the chance to launch the task locally. The master node also monitors the progress of the job execution. The slave nodes read the input blocks from a distributed storage system and apply the user-defined functions on them. Next, we present how Spark provides fault tolerance guarantees.

Fault tolerance in Spark. RDDs provide an interface that allows users to apply coarse-grained transformations on them. These transformations are persistently stored as lineage information. If failure happens, Spark will re-construct the lost data (i.e., RDDs on the failed nodes) by applying transformations on the corresponding input data in the persistent distributed storage system with the help of the lineage information. This obviates the need for data replication (for the intermediate output data) – as in Hadoop – with the extra cost of the re-computing the lost data sets.

More examples of widely adopted Big Data processing frameworks. Here, we briefly present the other popular Big Data processing frameworks that are inspired by the success of MapReduce programming model:

Flink. With the increasing need for fast processing due to the data generation at high velocities, Flink has been introduced to provide fast data processing (i.e., stream data processing). Flink is a distributed dataflow streaming engine which aims at providing fault-tolerant and efficient stream processing. Besides stream processing, Flink also provides support for batch processing, graph processing and machine learning applications.

Storm. Storm is a scalable and fault-tolerant real-time data processing framework. Similarly to Flink, Storm performs data processing in real-time and thus tries to cope with the high data velocities. Storm can process unbounded streams of data and thus provides a support for wide range of applications including online machine learning, ETL and continuous computation.

GraphLab. GraphLab [50] is a graph-based, efficient data processing framework which specifically targets machine learning applications. GraphLab provides a graph-based data model to address the limitation of MapReduce model in processing the data efficiently when there are high number of computational dependencies among the data.

To cover wider range of Big Data applications, recently several other data processing frameworks and tools have been introduced. Some examples include: (i) Apex [8], Samza [112], Heron [81], Beam [14] and Kafka [77] for supporting stream data processing; (ii) Giraph [47], GraphX [142] and Gelly [23] for graph processing; (iii) Hive [128], Spark-SQL [10] and BlinkDB [2] for performing query analytics.

2.1.4 Big Data Processing in Large-Scale Shared Platforms: Early Adoptions

Big Data Processing in Clouds

Cloud platforms facilitate Big Data processing at large-scale by providing various resources (e.g., compute, storage, network) to the users on-demand, by the virtue of virtualization [9,

22, 75]. Users can obtain these resources on a pay-as-you-go basis [96]. This makes clouds an easy to use, flexible and economically viable platform for performing Big Data processing.

Given the convenience of clouds as data processing platforms, industry and research labs have been trying to leverage them for performing efficient Big Data processing. In particular, there were many efforts focusing on the key aspects of MapReduce programming model including data locality and fault tolerance. For instance, several efforts have been conducted in order to achieve a better data locality thus improving the application performance [6, 63, 64]. Moreover, many studies tried to minimize the impact of failures on the performance of Big Data applications [32, 33, 107]. Although these studies can improve the application performance, they are applicable for single job scenarios only.

It is quite common to share the clouds among multiple users and applications. To efficiently operate shared clouds towards certain objectives (e.g., performance, fairness), Hadoop is equipped with several state-of-the-art schedulers:

The Fifo Scheduler. The first version of Hadoop comes with a fixed Fifo scheduler. In Fifo scheduling, the JobTracker simply pulls jobs from a single job queue. Although the scheduler's name suggests the prioritization of earlier submitted jobs, Fifo scheduler also takes into account jobs' priority. For data locality and fault tolerance, Fifo scheduler relies on the Hadoop core and thus it tries to achieve these aspects as we explained in Section 2.1.3.

The Fair Scheduler. Hadoop is also augmented with the Fair scheduler for ensuring fairness among multiple jobs in shared Hadoop clusters. The Fair scheduler assigns resources to jobs in a way such that, on average over time, each job gets an equal share of the cluster's resources. Short jobs are able to access the cluster resources, and will finish intermixed with the execution of long jobs. The Fair scheduler is primarily developed by Facebook, and aims at providing better responsiveness for short jobs, which are the majority at Facebook production clusters [150].

Beyond these state-of-the-art schedulers, many solutions have been proposed in order to facilitate running multiple applications on shared clouds with different fairness policies [3, 45, 46, 70, 87, 99, 104, 124, 140, 150] and priority requirements [27, 28, 78, 103, 113]. Some of these works have been targeting I/O latency by aiming at high data locality [3, 150] but they still cope with the failures in a best-effort way. To this end, we aim at sustaining high data locality while mitigating the impact of failures on the performance of Big Data applications with our Chronos scheduler in Chapter 6.

Big Data Processing in HPC Systems

HPC systems are known for providing high computing capability. They are equipped with high-speed networks, thousands of nodes with many cores and large memories [18, 80]. For instance, Sunway TaihuLight, No.1 in the top 500 supercomputers list, is a 10,649,600 processor supercomputer with a Linpack performance of 93 petaflop/s[1].

[1] http://www.top500.org/.

Given the high performance nature of HPC systems and the emergence of new applications with high performance requirements, we have recently seen several studies on leveraging these systems for Big Data processing [17, 42]. We can categorize these studies into four categories as following:

Characterizing the performance of Big Data applications on HPC systems. As first steps towards leveraging HPC systems for Big Data processing, several studies have been performed with the objective of understanding the performance of Big Data applications on these systems [25, 129, 139]. For instance, Wang et al. [139] performed an experimental study in order to investigate the characteristics of Big Data applications on a HPC system and this study demonstrates that I/O management plays an important role in performing efficient Big Data processing on HPC systems.

Employing MapReduce paradigm to process scientific applications. Recent studies investigate how to benefit from MapReduce paradigm to process scientific applications [39, 118, 141]. CGL-MapReduce [39] supports iterative computations and uses streaming for all the communications and applies MapReduce paradigm for High Energy Physics data analysis and k-means clustering of scientific data. MRAP [118] introduces scientific applications' access patterns into MapReduce paradigm in order to improve the MapReduce performance for scientific applications such as bioinformatics sequencing application.

Leveraging HPC technologies for Big Data processing. Several efforts have been conducted to incorporate HPC technologies (e.g., Infiniband networks, NVRAM and SSD as fast storage devices) into Big Data processing frameworks to improve the application performance. In Hadoop, communication among the machines is realized through remote procedure calls based on Java sockets which can create a performance bottleneck for Big Data applications. To alleviate this communication bottleneck of Hadoop, Lu et al. [91] tried to leverage Infiniband network technology. In particular, they provided a Remote Direct Memory Access (RDMA) support for Hadoop that enables high performance communication. They also benefited from Infiniband networks for other components in Big Data stack such as HDFS [71] and HBase [61]. Some works also tried to utilize fast storage devices to enhance the performance of Big Data applications. For example, Islam et al. [73] proposed a hybrid design consisting of NVRAMs and SSDs, as an alternative to disks, for data storage in a MapReduce cluster.

Introducing novel Big Data processing frameworks targeting HPC architectures. New implementations of MapReduce programming model — beyond Hadoop — have been introduced which are specifically tailored for HPC architecture such as multi-core CPU and GPU [40, 57, 148]. For instance, Phoenix is a novel MapReduce implementation targeting multi-core machines. In Phoenix, shared-memory threads are used for map and reduce functions. Phoenix also inspired many other works [56, 74, 94] that target multi-core machines. Mars [56] is a MapReduce framework on GPUs that uses a large number of GPU threads for map and reduce functions. However, Mars and Phoenix or other similar works are not suited for large volumes of data since they run on a single machine only.

As Big Data processing on HPC systems has been becoming a reality, the aforementioned studies are indeed important as pioneers towards this goal. However, none of these studies

fully address the challenges regarding the I/O management (e.g., latency and interference) which play a crucial role in enabling efficient Big Data processing on HPC systems. Next, we focus on I/O management in large-scale shared platforms and the respective challenges in detail.

2.2 I/O Management in Large-Scale Shared Platforms

In the recent years, we observe major advancements in the computation capabilities of data processing platforms in response to the unprecedented growth in the data sizes. However, the increase in the I/O capabilities of these platforms is not at the same level. For instance, Los Alamos National Laboratory moved to Trinity supercomputer from Cielo in 2015. While Cielo provided a peak performance of 1.37 Petaflops with a peak I/O throughput of 160 GB/s, its successor Trinity has a peak performance of more than 40 Petaflops for a I/O throughput of 1.45 TB/s [130]. This points out the 28-fold improvement in the computation capabilities with only a 9-fold increase in the I/O throughput. With such a big gap between the computation and I/O capabilities, I/O management solutions become critical in performing efficient Big Data processing on these platforms.

2.2.1 I/O Latency

One major challenge that needs to be tackled for efficient Big Data processing is the I/O latency problem. Data movements in the network is a well-known source of overhead in today's large-scale platforms. For example, Sunway TaihuLight (ranked 1st in the Top 500 list of supercomputers [2]) can only reach 0.37 Petaflops of performance on HPCG benchmark [34] due to the high number of data movements while its peak performance is 93 Petaflops [11]. Moreover, this problem will be more significant as the generated amount of the data is increasing everyday.

Mitigating I/O latency in clouds. Given the cost of data movements for the application performance, data locality has become a major focus in Big Data processing. Therefore, state-of-the-art scheduling strategies (i.e., Fifo, Fair, Capacity) in Big Data systems are mainly accommodated with locality-oriented strategies in order to sustain high data locality. Moreover, a large body of studies have sought to reduce I/O latency in the Big Data context by further improving the data locality [6, 63, 64, 131, 134, 149, 150]. Zaharia et al. introduced a delay scheduler [150], a simple delay algorithm on top of the default Hadoop Fair scheduler. Delay scheduling leverages the fact that the majority of the jobs in production clusters are short, therefore when a scheduled job is not able to launch a local task, it can wait for some time to increase the chance to launch the task locally. Venkataraman et al. [134] have proposed KMN, a MapReduce scheduler that focuses on applications with input choices and exploits these choices for performing data-aware scheduling. This in turn results in less congested links and better performance. Ananthanarayanan et al. introduced Scarlett [6], a system that employs a popularity-based data replication approach to prevent machines that store popular content from becoming bottlenecks, and to maximize locality. Yu et al. [149] proposed to use data prefetching to improve the application performance by mitigating the

[2] http://www.top500.org/.

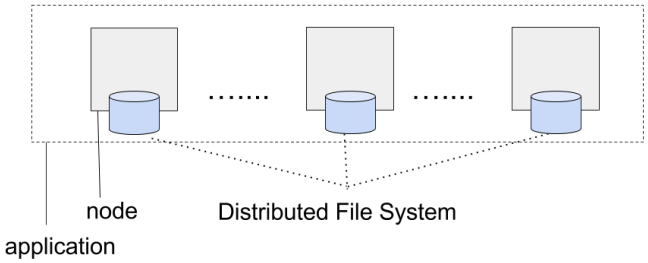
slowdown that is caused by the tasks with non-local data. Ibrahim et al. have proposed Maestro [63], a replica-aware map scheduler that tries to increase locality in the map phase and also yields better balanced intermediate data distribution. We address data locality problem in Chapter 6 where we present a failure-aware scheduler that aims at sustaining a high data locality even under failures.

Mitigating I/O latency in HPC systems. Aforementioned studies mainly target clouds. However, these studies would not be appropriate for HPC systems where the input data typically resides in a parallel file system rather than the local storage of the machines. Hence, all the machines have an equivalent distance to the input data in HPC systems. Consequently, we recently observed architecture-based solutions (e.g., data staging nodes, burst buffers) in order to reduce the I/O latency in HPC systems. Burst buffer (BB) is an intermediate storage layer between the parallel file system and computation nodes which consists of high throughput storage devices such as SSD, RAMDisk and NVRAM. Liu et al. [88] demonstrated that BBs can minimize the time spent in the I/O by scientific applications by acting as a data staging area and allowing to hide the high costs of data transfers in the network. In [15], the authors extended PLFS, a middleware virtual file system, with burst buffer support which helped to improve the performance of scientific applications by 30%. In [138], burst buffers brought a 8-fold improvement to the I/O performance of scientific applications by buffering the bursty I/O (i.e., checkpointing) of scientific applications and eventually flushing this data to the parallel file system. The use of dedicated I/O resources [35, 156] also resemble to burst buffer solution. These I/O resources help to hide the I/O latency by transferring the data to these resources asynchronously and thus allow to overlap computation with I/O.

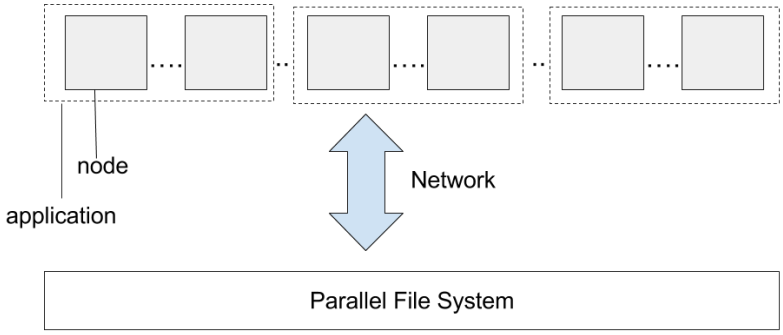
The main concept underlying the design of burst buffers is the fact that scientific applications have a periodical behavior. These applications perform bursty I/O requests with a defined time period. Moreover, this bursty I/O behavior dominates the I/O traffic in HPC systems [137]. Thus, burst buffer is a great fit for the scientific applications. On the contrary, leveraging burst buffers for Big Data processing in order to mitigate I/O latency problem is challenging given that Big Data applications mostly run in batches thus requires a continuous interaction with the parallel file system for reading the input data. We address this challenge in Chapter 5 where we present a burst buffer solution that helps to improve the performance of Big Data applications in HPC systems.

2.2.2 I/O Interference

Another major challenge towards efficient Big Data processing on large-scale shared platforms is I/O interference. In the context of HPC systems, different from clouds, the storage system (often present as a centralized set of storage servers) is shared among multiple applications running on the computation nodes as illustrated in Figure 2.2. Thus, sharing the storage system leads to a performance degradation for these concurrent applications. This performance degradation resulting from I/O interference is often represented with an *interference factor* metric. The interference factor is defined as the ratio between the I/O time of an application when it is contending with other applications for its I/O operations and the time for it to perform the same I/O operations alone.



(a) Typical storage system in clouds



(b) Typical storage system in HPC Systems

Figure 2.2: Typical storage systems in clouds and HPC systems.

$$I = \frac{T}{T_{alone}} > 1 \tag{2.1}$$

It is also important to note that the ratio among computation nodes and storage servers is usually between 10x and 100x [97]. These uncontrolled I/O performance degradations in large-scale applications lead to a large waste of computation time since applications will be idle while waiting for their I/O operations to be completed. Therefore, mitigating the I/O interference problem is necessary to prevent I/O from being bottleneck to the performance of Big Data applications.

To this end, researchers have been trying to handle the I/O interference problem in HPC systems. The causes of I/O interference can be very diverse. Some works focus on finding solutions at the disk level [154, 155] mainly by trying to optimize the access locality. Some research efforts consider network contention as the major contributor to the I/O interference [13, 16, 76, 83, 106, 125]. Use of interference-aware schedulers [37, 43, 121, 153, 157] can

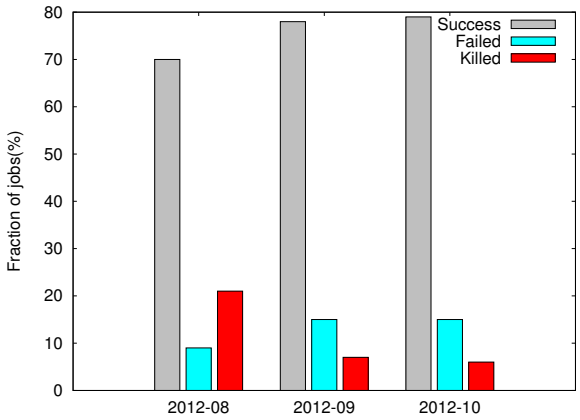


Figure 2.3: Job statistics of a Hadoop cluster in Carnegie Mellon University.

help to control the level of interference by mainly coordinating the requests from different applications. Towards interference-aware Big Data processing in HPC systems, we try to identify the root causes of I/O interference and look at the interplay between these causes in Chapter 4.

Moreover, this interference issue can even become more severe when scientific and Big Data applications are co-located on the same platform (i.e., sharing the same storage system). In Chapter 5, we address this problem and try to minimize the performance degradation resulting from the I/O interference between HPC and Big Data applications.

2.3 Handling Failures at Large Scale

Given the scale of Big Data processing platforms, failures are a norm rather than being an exception. For example, BlueWaters supercomputer at the National Center of Supercomputing [18] has a mean time to failure of 4.2 hours [31]. In particular, clouds consist of hundreds of failure-prone commodity machines in order to provide efficient data processing while enabling low cost of implementation for the users. Moreover, these platforms are shared by multiple users with different application characteristics which significantly increases the probability of failures. Therefore, it is critical to efficiently handle failures for performing efficient Big Data processing. We address this challenge in Chapter 6 by trying to enable fast failure recovery in shared Hadoop clusters. Hereafter, we briefly list the failure handling strategies in Big Data processing and then we present the current efforts on mitigating the impact of the failures on the performance of Big Data applications in Hadoop clusters.

2.3.1 Failure-Handling Strategies in Big Data Processing

Failures are part of the Big Data processing frameworks' characteristics [32]. Even worse, they can result in a high number of failed jobs. Figure 2.3 shows the job statistics of a Hadoop cluster in the Carnegie Mellon University [100] for the period of three months in 2012. Fraction of the failed jobs (i.e., 13% on average) demonstrates that failures are part of the Big

Data processing clusters. Hence, several strategies have been introduced in order to cope with failures:

Replication [29, 66] ensures data availability by creating extra copies of the same data in different machines. If some failure happens, the tasks of the failed machine can be re-executed on other healthy machines which have the copy of the tasks' data. Replication is one of the most commonly used strategy in Big Data processing frameworks. For example, Hadoop framework (on which we focus in Chapter 6) handles machine failures (i.e., fail-stop failure) by re-executing all the tasks of the failed machines by leveraging data replication as well.

Checkpointing [92, 98, 105] stores the snapshot of the state of a task at defined time intervals. Unlike replication, the task can be restarted from the latest checkpoint rather than from the beginning when failure happens. However, this strategy can be costly when the amount of data to checkpoint is large. For example, DataMPI [92], communication library that extends MPI for supporting Big Data analytics, uses checkpointing as a fault-tolerance strategy and this strategy incurs a significant overhead: 12% performance loss when the checkpointing is enabled.

Lineage [84, 152] stores the changes (e.g., transformations) performed on the data as a lineage graph and uses this graph to re-compute the lost data in case of failures. The lineage mechanism is originally used in the database domain [21] and recently Spark [152] adopted lineage as its fault-tolerance mechanism.

2.3.2 Mitigating Failures in Shared Hadoop Clusters

Single job. Failures can significantly degrade the performance of Big Data applications in Hadoop clusters. For instance, Dinu et al. [32] have demonstrated a large variation in the performance of Big Data applications in the presence of failures. In particular, they reported that the performance of a Big Data application degrades by up to 3.6x for one single machine failure. Therefore, several studies have been dedicated to explore and improve the performance of Big Data applications under failures. Ruiz et al. have proposed RAFT [107], a family of fast recovery algorithms upon failures. RAFT introduces checkpointing algorithms to preserve the work upon failures. RAFT was evaluated using single job only. Dinu et al. have proposed RCMP as a first-order failure resilience strategy instead of data replication [33]. RCMP performs efficient job recomputation upon failures by only recomputing the necessary tasks. However, RCMP only focuses on I/O intensive pipelined jobs, which makes their contribution valid for a small subset of Big Data applications.

Moving towards shared Hadoop clusters with multiple concurrent jobs. Failure handling and recovery has long been an important goal in Hadoop clusters. However, previous efforts to handle failures either have been targeting single job scenarios in Hadoop clusters or entirely entrusted to the core of Hadoop as we mentioned in Section 2.1.4 regarding the state-of-the-art Hadoop schedulers. The unawareness of failures may therefore prevent Hadoop schedulers from operating correctly towards meeting their objectives (e.g., fairness, job priority) and can significantly impact the performance of Big Data applications. To this

end, we introduce a failure-aware scheduler for shared Hadoop clusters in Chapter 6. We also aim at sustaining a high data locality even under failures.

2.4 Discussion: Enabling Efficient Big Data Processing for Large-Scale Shared Platforms

Big Data applications are a driving source in the advancements for our society by turning data into a valuable information. One main challenge in this process, consists of dealing with the large amounts of data generated by these applications. To meet this challenge, data processing platforms become larger and larger, and these large-scale platforms are being used by more applications at the same time.

Given the scale and shared nature of these platforms, I/O management and failure handling solutions have become crucial in enabling efficient data processing on these platforms. In this thesis, we take *application performance* as an objective regarding the efficiency of data processing. One major problem is the I/O latency where movements of huge amounts of data in the network can impose a large overhead for the performance of Big Data applications. As these platforms are shared by a high number of concurrent applications, I/O interference also appears as a major performance bottleneck. Novel and smart I/O management solutions are needed, alongside these powerful data processing platforms.

Another major source of performance degradation in Big Data processing is failures. Due to the large scale and shared nature of the data processing platforms, failures are an everyday reality rather than being an exception. Handling failures therefore is a key to ensure efficient Big Data processing on these platforms.

In the next chapters, we describe through a number of contributions how we address the aforementioned challenges regarding efficient Big Data processing on large-scale shared platforms.

Chapter 3

Characterizing the Performance of Big Data Applications in HPC Systems

Contents

3.1 Motivation . . . 22

3.2 Performance Bottlenecks of Big Data Applications on HPC Systems . . . 23

3.2.1 Methodology . . . 23

3.2.2 The Impact of I/O Latency . . . 24

3.2.3 The Impact of Contention . . . 29

3.2.4 The Impact of the File System Configuration . . . 31

3.3 Toward Performing Efficient Big Data Processing on HPC Systems . . . 32

3.4 Related Work . . . 34

3.5 Conclusions . . . 35

With the recent interest towards leveraging HPC systems for Big Data processing, it becomes important to identify the performance characteristics of Big Data applications on these systems. This is important since Big Data applications are mainly optimized to run on clouds where main objective is to process data rather than computation as in HPC systems. To this end, we perform an experimental study characterizing the performance of Big Data applications on HPC systems. We use representative Big Data workloads on the Grid'5000 [19] testbed to evaluate how latency, contention, and file system's configuration impact the performance of Big Data applications on HPC systems.

3.1 Motivation

We have recently seen a huge interest in leveraging HPC systems for Big Data processing [17, 42]. However, when doing this one should be aware of the different architectural designs in current Big Data processing and HPC systems. Big Data processing clusters have shared nothing architecture (e.g., nodes with individual disks attached) and thus they can co-locate the data and compute resources on the same machine (i.e., data-centric paradigm). On the other hand, HPC clusters employ a shared architecture (e.g., parallel file systems) [51] which results in separation of data resources from the compute nodes (i.e., compute-centric paradigm). These differences in the design of these two systems lead us to the following questions that we try to answer in this chapter:

How does I/O latency impact the performance of Big Data applications? Big Data systems are mainly optimized to sustain high *data locality* to reduce the I/O latency which is the major contributing factor to the application performance [6, 134]. However, the same approach on the data locality can not be applied in HPC systems since all the nodes have an equivalent distance to the input data. In addition, employing a shared parallel file system also results in remote data transfers through network when Big Data applications performing I/O thus resulting in a *higher latency*.

Previously, several studies have been conducted to explore the impact of the I/O latency in HPC storage architecture on the performance of Big Data applications. For instance, Wang et al. [139] indicated that having the storage space as Lustre file system [117] (compared to co-locating it with the compute nodes) can lead to 5.7x performance degradation for Big Data applications. Similarly, one of our objectives is to provide a detailed analysis of the impact of latency on the performance of Big Data applications by considering the different phases of these applications as input, intermediate and output data.

What is the impact of I/O contention on the performance of Big Data applications? *I/O contention* is a well-known source of performance degradation for HPC applications due to the large size and shared architecture of HPC clusters [43, 89]. If HPC systems are to serve as the underlying infrastructure for Big Data processing, the impact of contention will be amplified given the huge data sizes of these applications.

Unfortunately, previous studies regarding the characterization of Big Data applications on HPC systems [25, 129, 139] do not quantitatively analyze the impact of contention on the performance of Big Data applications despite its importance.

What is the impact of the file system specific properties on the application performance? Parallel file systems have several specific properties to meet the requirements of HPC applications. For instance, synchronization feature in PVFS provides resiliency for HPC applications. These properties can have a significant impact on the performance of Big Data applications. For example, Chaimov et al. [25] found out that metadata operations in Lustre file system create a performance bottleneck for Big Data applications. Similarly, we aim to explore the potential performance issues specific to PVFS.

How can we benefit from (HPC) architecture-driven I/O management solutions for performing efficient Big Data processing? Our last objective is to discuss the implications of the findings with respect to the aforementioned questions and draw attention to new ways (e.g., burst buffers) which can alleviate the performance issues resulting from HPC storage architecture and thus improve the performance of Big Data applications on HPC systems.

3.2 Performance Bottlenecks of Big Data Applications on HPC Systems

3.2.1 Methodology

We conducted a series of experiments in order to assess the impact of the potential issues regarding HPC systems (i.e., latency, contention, file system's configuration) on the performance of Big Data applications. We further describe the experimental environment: the platform, deployment setup, and Big Data workloads.

Platform Description

The experiments were carried out on the Grid'5000 testbed. We used the Rennes site; more specifically we employed nodes belonging to the *parasilo* and *paravance* clusters. The nodes in these clusters are outfitted with two 8-core Intel Xeon 2.4 GHz CPUs and 128 GB of RAM. We leveraged the 10 Gbps Ethernet network that connects all nodes of these two clusters. Grid'5000 allows us to create an isolated environment in order to have full control over the experiments.

Deployment Setup

We used Spark version 1.6.1 working with HDFS version 1.2. We configured and deployed a Spark cluster using 51 nodes on the *paravance* cluster. One node consists of the Spark master and the HDFS NameNode, leaving 50 nodes to serve as both slaves of Spark and DataNodes. We used the default value (number of available cores on the node) for the number of cores to use per each node. Therefore, the Spark cluster can allocate up to 800 tasks. We allocated 24 GB per node for the Spark instance and set Spark's default parallelism parameter (spark.default.parallelism) to 800 which refers to the number of RDD partitions (i.e., number of reducers for batch jobs). At the level of HDFS, we used a chunk size of 32 MB and set a replication factor of 2 for the input and output data.

The OrangeFS file system (a branch of PVFS2 [110]) version 2.8.3 was deployed on 12 nodes of the *parasilo* cluster. We set the stripe size which defines the data distribution policy in PVFS (i.e., analogous to block size in HDFS) to 32 MB in order to have a fair comparison with HDFS. Unless otherwise specified, we disabled the synchronization for PVFS (Sync OFF) which indicates that the incoming data can stay in kernel-provided buffers. We opted for Sync OFF configuration since Spark is also using the memory as a first storage level with HDFS.

Workloads

We selected three representative Big Data workloads including Sort, Wordcount and PageRank which are part of HiBench [58], a Big Data benchmarking suite.

Wordcount is a map-heavy workload which counts the number of occurrences of each word in a data set. The map function splits the input data set into words and produces the intermediate data for the reduce function as a key,value pair with word being the key and 1 as the value to indicate the occurrence of the word. The reduce function sums up these intermediate results and outputs the final word counts. Wordcount has a light reduce phase due to the small amount of the intermediate data.

Sort is a reduce-heavy workload with a large amount of intermediate data. This workload sorts the data set and both map and reduce functions are simple functions which take the input and produces its sorted version based on the key. This workload has a heavy shuffling in the reduce phase due to the large amount of intermediate data it produces.

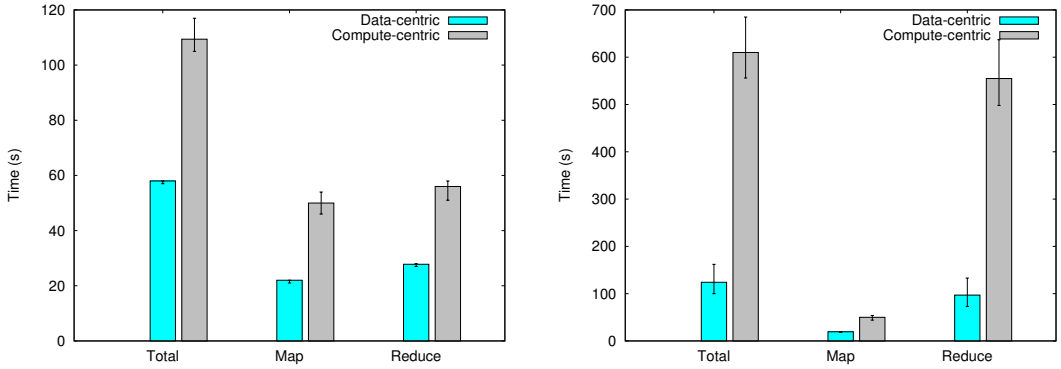
PageRank is a graph algorithm which ranks elements according to the number of links. This workload updates these rank values in multiple iterations until they converge and therefore it represents the iterative set of applications. For Sort and Wordcount workloads, we used 200 GB input data set generated with RandomTextWriter in HiBench suite. For the PageRank workload, we also used HiBench suite which uses the data generated from Web data with 25 million edges as an input data set.

3.2.2 The Impact of I/O Latency

First, we try to understand the impact of the data location on the application performance. While storage resources are co-located with Spark tasks under the data-centric paradigm (i.e., when using Spark with HDFS), Spark tasks need to communicate with the parallel file system either to fetch the input data or to write the output data under the compute-centric paradigm (i.e., when PVFS is employed with Spark). This remote data access results in a higher latency compared to the data-centric paradigm which leverages data locality. Figure 3.1 shows how latency can affect the application performance. Note that, intermediate data is stored locally on the aforementioned settings for Spark in order to focus on the latency resulting from reading the input data in map phase. We explore the intermediate data storage separately in the next subsection.

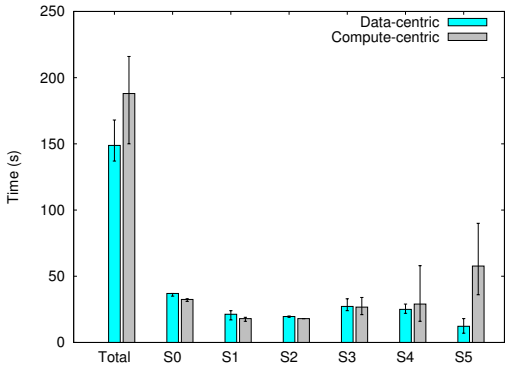
Figure 3.1(a) displays the execution time of the Wordcount workload for both paradigms with a performance in map and reduce phases. Overall, Wordcount performs 1.9x worse under the compute-centric paradigm compared to the data-centric one. When we look at the performance in each phase, we observe that the performance degradation contributed by the map phase (2.3x) is higher compared to the reduce phase. This stems from the fact that Wordcount has a light reduce phase and generates only a small amount of output data.

Similarly, in Figure 3.1(b) we observe that the data-centric configuration outperforms the compute-centric one by 4.9x for the Sort workload. In contrast to Wordcount, the reduce phase is the major contributor to the performance degradation. For the Sort workload, the amount of the output data is equal to the input data thus it suffers from a higher latency in the reduce phase as data is written to the parallel file system. As a result, having a higher



(a) Wordcount.

(b) Sort.



(c) PageRank.

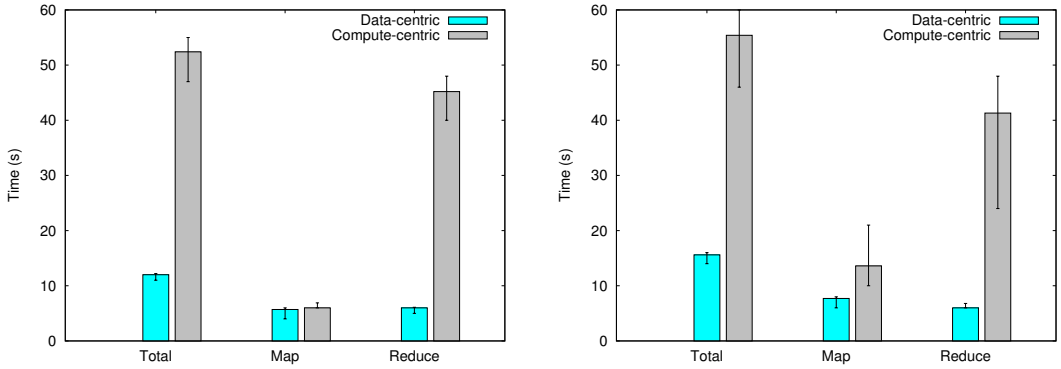
Figure 3.1: Performance of Big Data workloads on Spark under data-centric and compute-centric paradigms.

latency on both input and output phases led to higher performance degradation for the compute-centric paradigm.

Lastly, we ran the PageRank workload in both settings for Spark and Figure 3.1(c) shows the results. Here, performance degradation with the compute-centric paradigm is only 26%. The reason behind this is that I/O phases of the PageRank workload (i.e., Stage 0 and Stage 5 (denoted as S0 and S5)) accounts for a small fraction of PageRank execution time and Spark computes the iterations (i.e., Stage 1, 2, 3 and 4) locally.

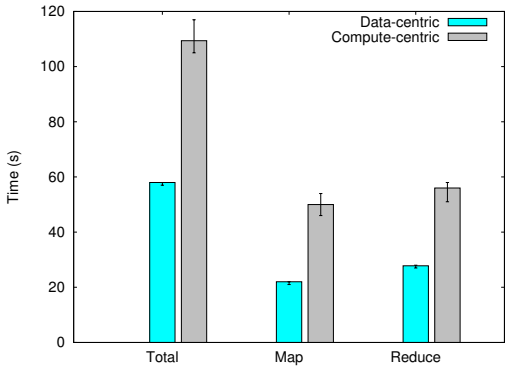
The Impact of the Input Data Sizes

We also investigated the impact of the input data size on the application performance. To do so, we ran the Wordcount workload with different input sizes as 2 GB, 20 GB and 200 GB. Figure 3.2 displays the performance of the Wordcount workload in each phase for data and compute-centric paradigms. Overall, we observe that the impact of I/O latency is only visible in the map phase for the compute-centric paradigm with increasing input sizes: there is a performance degradation for the map phase by 1.2x, 1.8x and 2.3x with 2 GB, 20 GB and 200 GB input sizes, respectively. This is mainly due to the fact that Wordcount is a map-



(a) 2 GB.

(b) 20 GB.



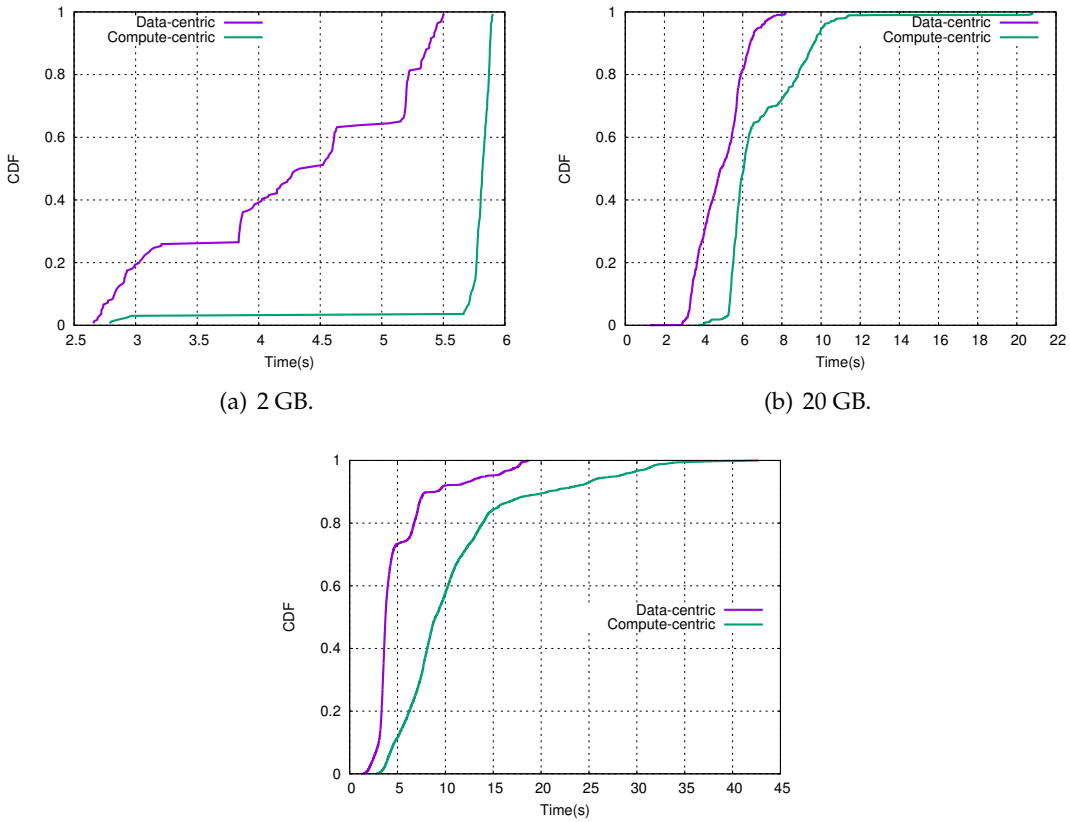
(c) 200 GB.

Figure 3.2: Performance of the Wordcount workload with different input sizes.

heavy workload which generates a small amount of output data and therefore reduce phase results do not vary significantly with respect to different data sizes. To further investigate these different behaviors in map and reduce phases, we display the CDF of map and reduce task durations in Figures 3.3 and 3.4.

Interestingly, Figure 3.3(a) shows that some map task durations are smaller for the compute-centric paradigm compared to the data-centric one. This is due to the fact that Spark employs delay scheduling [150] to increase the chances of a map task to be launched locally for the data-centric paradigm. This delay while launching the map tasks, which results in a performance degradation for the jobs with small input data sizes, is due to the default Spark configuration for the maximum waiting time (i.e., 3 seconds) in scheduling the map tasks. This is only valid for the data-centric paradigm since there is no data locality objective when scheduling the tasks in the compute-centric paradigm where all the machines have an equivalent distance to the parallel file system. On the other hand, we observe an increase in the map task durations with larger input sizes for the compute-centric paradigm. This results from the higher latency while fetching the input data from parallel file system with larger input sizes.

Another interesting trend we observe is that the maximum map task duration also increases with the increasing data sizes, especially with 200 GB input data size in Figure 3.3(c).



(a) 2 GB.

(b) 20 GB.

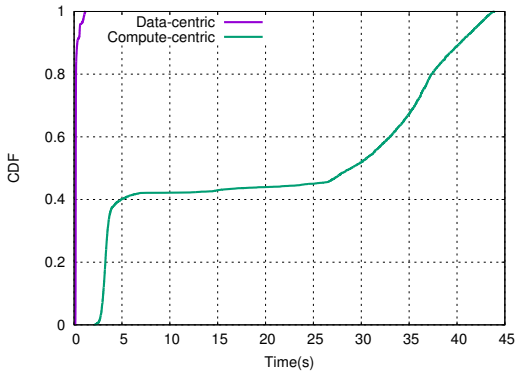
(c) 200 GB.

Figure 3.3: CDFs of running times of map tasks in the Wordcount workload.

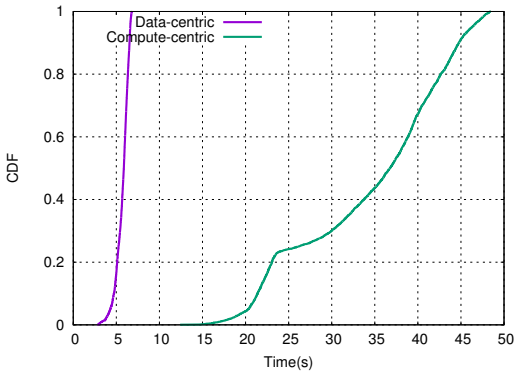
We believe that this behavior is due to the higher contention with the increased number of concurrent map tasks. It is important to note that there are 33, 594 and 800 concurrent map tasks with 2 GB, 20 GB and 200 GB input sizes. Moreover, we see that this increase is much higher with the compute-centric paradigm which can highlight the severity of the contention problem for this paradigm. We will further explain the impact of the contention on the application performance in Section 3.2.3.

In Figure 3.4, we observe a similar trend for the reduce task durations for the compute-centric paradigm. With larger data sizes, we observe an increase in those durations too. This again stems from an increased amount of the remote data transfer while writing the reducer outputs to the parallel file system. Moreover, we discover that there is a high performance variability in the reduce phase and the maximum task duration is quite high even with 2 GB data size. This is due to the static Spark configuration which employs 800 reducers regardless of the input data size. These high number of reducers overload the parallel file system and results in this performance variability. Hence, we do not see the impact of latency in Figure 3.2 for the reduce phase. However, when the output data size is large enough as shown for the Sort workload in the previous experiment (Figure 3.1(b)), the impact of the I/O latency is quite clear as it results in a significant performance degradation.

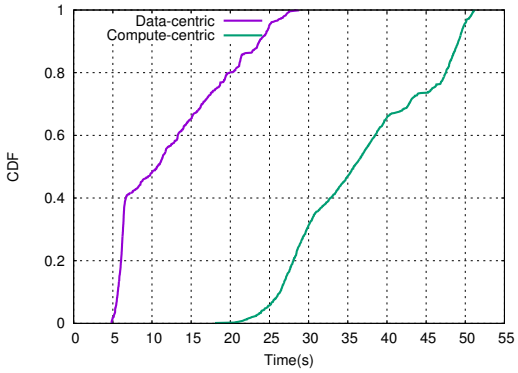
For the data-centric paradigm, this time we see that reduce task durations are inlined



(a) 2 GB.



(b) 20 GB.



(c) 200 GB.

Figure 3.4: CDFs of running times of reduce tasks in the Wordcount workload.

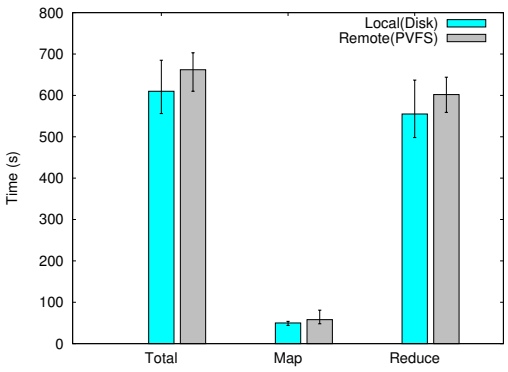


Figure 3.5: Impact of the location of intermediate data on the performance of the Sort workload.

with the data sizes, different from the map phase. While for the map phase there is an increase in the maximum task duration due to the increased number of concurrent map tasks, for the reduce phase the number of reduce tasks is fixed and the increase in the reduce task durations is mainly due to the increased amount of reducer output with larger input sizes.

Intermediate Data Storage

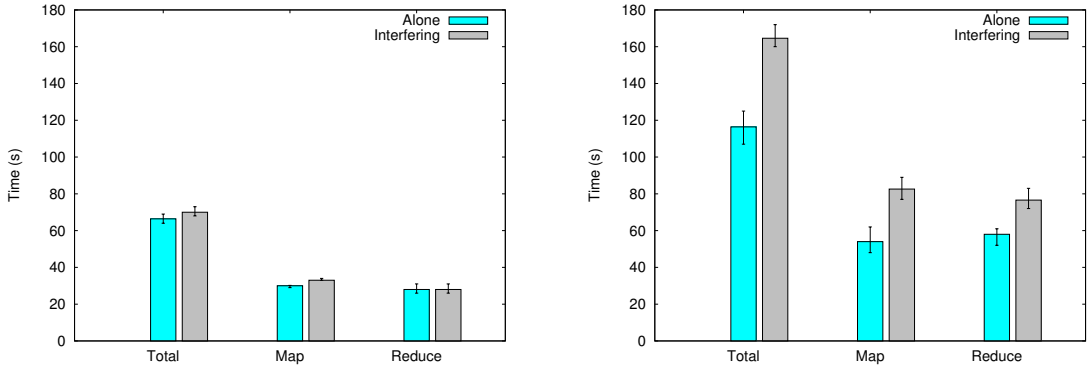
In Big Data processing systems, intermediate data are typically stored locally. However, nodes in some of the HPC systems may not have individual disks attached to themselves. This arises the question of how to store the intermediate data when running Big Data applications on HPC systems. As a naive solution, we employed PVFS also for storing the intermediate data as well as storage space for input and output data like in the experiments so far. We ran the Sort workload with PVFS since it generates an intermediate data equal to the input data size and thus it is a good fit to evaluate the intermediate data storage for HPC systems. Figure 3.5 compares the performance of Sort depending on the intermediate data location: local storage (on disk) or remote storage (on PVFS). We see that using PVFS also for the intermediate data storage space results in 9% performance degradation.

When we analyze the performance of the Sort workload in each phase, we see that this performance degradation is 16% for the map phase which stems from writing the intermediate data to the parallel file system. For the reduce phase, we observe that there is a 8% increase in the completion time due to the additional I/O latency when fetching the intermediate data from PVFS.

Finding: In all the workloads, we observe that the remote data access to the parallel file system leads to a significant performance degradation, especially for the input and output data. We also show that the degree of this performance degradation depends on the characteristics of the workloads and on the input data size.

3.2.3 The Impact of Contention

Given the shared architecture of HPC systems, contention is likely to occur when running Big Data applications on a HPC system. To assess the impact of contention on the performance of Big Data applications, we designed the following experiments:



(a) Performance of multiple Wordcount workloads under a data-centric paradigm.

(b) Performance of multiple Wordcount workloads under a compute-centric paradigm.

Figure 3.6: Performance of multiple Wordcount workloads under different paradigms.

Measuring the contention when running concurrent Big Data applications. Since the storage system is shared by all the nodes, this can create a serious contention problem on the storage path including network, server and storage devices. Here, we ran two Wordcount workloads concurrently under compute and date-centric paradigms by employing the Fair scheduler in Spark. The Fair scheduler allows these workloads to have equal share of the resources in the Spark cluster (i.e., each workload employ 400 tasks which is equal to the half of the cluster capacity). Figure 3.6 displays the execution times of the Wordcount workload when it runs alone and together with the other Wordcount workload for data and compute-centric paradigms. As shown in Figure 3.6(a), the performance degradation when running in contention with the other Wordcount workload is negligible with data-centric paradigm. In contrast, we observe that there is a 41% performance degradation with the compute-centric paradigm when two workloads are running concurrently. This stems from sharing the same parallel file system with compute-centric paradigm while these two workloads perform their I/O operations on their individual storage devices in the data-centric paradigm. In particular, Figure 3.6(b) highlights that this performance degradation is mainly due to the map phase. This is because Wordcount is a map-heavy workload and therefore the number of I/O operations is quite large in the map phase compared to the reduce phase.

Measuring the contention when co-locating HPC and Big Data applications. This contention problem can even become more significant when we consider the ultimate objective of the HPC and Big Data convergence which is co-locating scientific and Big Data applications on a same platform. To emulate this objective, we ran the Wordcount workload alone and together with the IOR workload. IOR [119] is a popular I/O benchmark that allows users to specify different I/O configurations and thus measures the I/O performance of HPC systems. For IOR workload, we employed 224 processes (on a different set of nodes separated from the ones running the Wordcount workload) where each process issues a 512 MB write request in 32 MBs of chunks. Figure 3.7 shows the execution times of the Wordcount workload for both cases. Due to resource sharing (file system and network) with the IOR workload, there is a 1.4x performance degradation in the total execution time of the Wordcount workload. When we look at the performance in each phase, we observe that this

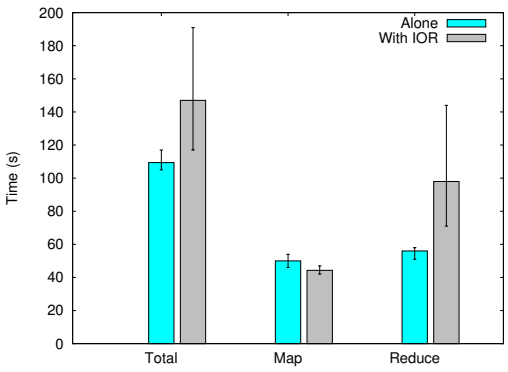


Figure 3.7: Performance of the Wordcount workload when running alone and together with IOR workload.

performance degradation is mainly due to the reduce phase. This stems from the fact that reduce phase performs write operations as the IOR workload and this results in a write/write contention.

Finding: We demonstrate that contention appears as a limiting factor for Big Data applications on HPC systems due to employing a shared storage system. In Chapter 4, we further investigate the causes of this problem and try to understand its implications on I/O management solutions for performing efficient Big Data processing on HPC systems.

3.2.4 The Impact of the File System Configuration

Besides the generic problems of HPC systems as latency and contention, we can also encounter performance issues with the file system specific problems when running Big Data applications on HPC systems. For example, [25] reported that metadata operations on Lustre create a bottleneck for Spark applications. Thus, we wanted to investigate file system specific problems that Spark applications can encounter. To this end, we configured PVFS with synchronization enabled (Sync ON). This synchronization feature can be necessary for providing a better reliability guarantee for the clients. To ensure this, each request is immediately flushed to the disk before finalizing the request.

We ran the Wordcount workload with two different synchronization options for PVFS: Sync ON and Sync OFF. Figure 3.8 shows that Wordcount performs 1.5x worse when synchronization is enabled. We observe that this significant performance degradation mainly stems from the reduce phase. This is expected since the output data is sent to the file system during the reduce phase and each request is flushed to the disk thus resulting in a major bottleneck for the application performance.

We also ran the Sort workload with two different configurations of PVFS and Table 3.1 shows that Sort performs 4.5x worse when synchronization is enabled. In contrast to Wordcount, we observe a much higher performance degradation with the Sort workload. This is because while Sort is generating a large amount of output data (200 GB as the input data size), Wordcount has a light reduce phase and generates only a small amount of output data.

Finding: Parallel file systems are equipped with several features which are important for HPC applications (i.e., synchronization feature in PVFS to provide resiliency, dis-

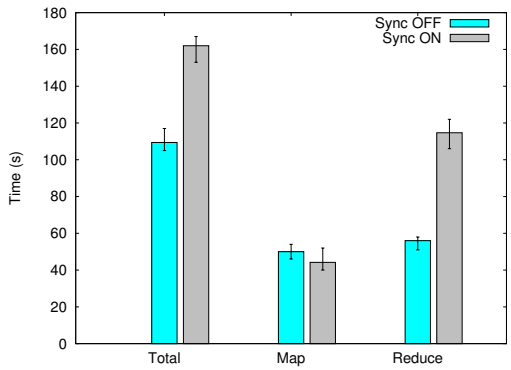


Figure 3.8: Performance of the Word-count workload under different configurations of PVFS.

Table 3.1: Execution time of the Sort workload and its phases under different configurations of PVFS.

Configuration	Execution Time	Map	Reduce
Sync ON	2708.5 s	42.7 s	2665.8 s
Sync OFF	597.6 s	42.6 s	555.0 s

tributed locking mechanism in Lustre to ensure file consistency). However, as reported earlier in [25, 139] and as demonstrated in our experiments, these features may bring a significant performance degradation for Big Data applications. We further investigate the role of these file system specific configurations on the I/O interference in Chapter 4.

3.3 Toward Performing Efficient Big Data Processing on HPC Systems

Our experiments revealed that Big Data applications encounter serious performance issues related to I/O management when running on HPC systems. On the other hand, current burst buffer solutions (i.e., intermediate storage layers with fast storage devices) [25, 72, 73, 88, 114] demonstrated their effectiveness in improving the I/O performance of HPC applications. These studies have focused on mitigating the latency resulting from writing and reading the intermediate data. However, our findings showed that using burst buffers for the intermediate data can bring an improvement of at most 9% when the intermediate data have the same size as the input data. As a result, the latency introduced by the intermediate data is not really the bottleneck for a major fraction of Big Data applications: by analyzing traces collected from three different research clusters we observe that the amount of the intermediate data is less than 20% of the input data size for 85% of the applications [52]. On the other hand, we find that the latencies resulting from reading the input data and writing the output data significantly impact performance. Thus, it is very important to mitigate the high latency resulting from accessing those data when developing burst buffer solutions which we address in Chapter 5.

On the Capacity and Location of Burst Buffers

Although burst buffers promise a large potential for Big Data applications, leveraging them efficiently is not trivial. For example, there is a trade-off between the capacity and the throughput of the storage devices that are used in the burst buffers. Although, storage devices such as SSDs or NVRAMs can provide high throughput, they are limited in the *storage capacity*. Moreover, we demonstrated in our experiments that we should tackle all the I/O phases (i.e., input, intermediate and output data) while addressing the latency problem. Therefore, the problem of having limited capacity will be amplified when we try to use the burst buffer for all the I/O phases. Similarly, it is important to decide when and which data to evict when running multiple concurrent applications.

Another challenge would be choosing the optimal *deployment location* for the burst buffers. Some of the possible deployment locations are within the compute nodes [139] or using a dedicated set of nodes [25] as burst buffers. While co-locating burst buffers and compute nodes can prevent the aforementioned capacity constraints since compute nodes are greater in size compared to dedicated nodes, this may result in a computational jitter due to sharing of the resources as also reported in [15].

To find out the impact of the aforementioned factors on the performance of Big Data applications, we emulated a naive adoption of burst buffers by using the *ramdisk* (i.e., /dev/shm/) as a storage space for the Big Data applications and performed the following experiments:

Measuring the impact of the storage capacity on the application performance. Here, we ran the Wordcount workload with two different storage capacities for the burst buffer as 40 GB and 10 GB memory. Note that, we used a smaller input data size than previous experiments which has a data size of 20 GB. The burst buffer is employing 5 dedicated nodes. Figure 3.9 shows the execution time of the Wordcount workload for different burst buffer configurations. We observe a 2.1x performance degradation when the burst buffer has 10 GB storage capacity. When we look at the performance of the workload in each phase, we see that this performance degradation is attributed to the map phase. This results from not having enough space for storing the input data on the burst buffer. Hence, compute nodes have to fetch the input data from the parallel file system thus resulting in a high I/O latency in the map phase. On the contrary, all I/O operations performed between burst buffer nodes and compute nodes when there is enough storage capacity. For the reduce phase, we do not observe any performance degradation since the output data to be written is small enough to fit into the burst buffer storage space, for this workload.

Measuring the impact of the deployment location of the burst buffer. We ran the Wordcount workload with the same configuration as in the previous experiment and deployed the burst buffer in two scenarios: in the first one, the burst buffer is deployed as a disjoint set of nodes and in the second one it is located as a subset of the compute cluster. Figure 3.10 displays that Wordcount performs better when burst buffer is deployed as a separate set of nodes. We hypothesize the following explanation. When the burst buffer is using the subset of the nodes of the compute cluster, I/O and compute tasks on those nodes conflict with each other thus resulting in a significant performance degradation (38% slowdown). This is in line with the observations reported in [15].

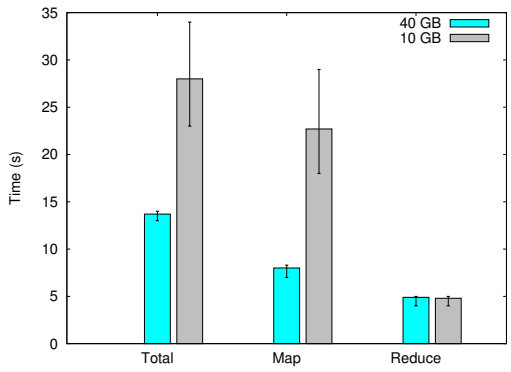


Figure 3.9: Impact of the memory capacity of the burst buffer on the performance of the Wordcount workload.

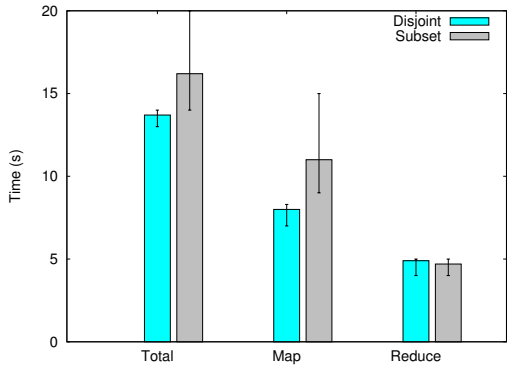


Figure 3.10: Impact of the location of the burst buffer on the performance of the Wordcount workload.

Finding: Our experiments show that the storage capacity and the location of burst buffers can have a significant impact on the performance of Big Data applications. With limited storage capacity, we demonstrate that burst buffers can not mitigate the latency problem fully since compute nodes still need to fetch most of the data from the parallel file system. For the deployment location, we observe that co-locating the burst buffer and compute resources on the same node can not be appropriate due to the possible interference among them. We take these findings into consideration when designing our burst buffer solution in Chapter 5 which tries to enable efficient Big Data processing on HPC systems.

3.4 Related Work

Several research efforts have been conducted to evaluate the performance of Big Data analytics frameworks on HPC systems. Wang et al. [139] performed an experimental study where they investigated the characteristics of Spark on a HPC system with a special focus on the impact of the storage architecture and locality-oriented task scheduling. Tous et al. [129] evaluated the Spark performance on a MareNostrum supercomputer. In particular, they studied the impact of different Spark configurations on the performance of Sort and K-means applications. In [126], the authors compared the performance of MapReduce applications on PVFS and HDFS file systems by using Hadoop framework and give insights into how to emulate HDFS behavior by using PVFS. Li et al. [85] compared the performance of MapReduce applications on scale-up and scale-out clusters and proposed a hybrid scale-

up/out Hadoop architecture based on their findings.

The aforementioned studies provide useful findings towards leveraging HPC systems for Big Data processing. However, they do not illustrate a complete analysis of the potential performance issues (e.g., latency and contention). For the latency problem, most of the studies focus on the intermediate data storage and ignore the latencies which can occur in other I/O phases. We provide a detailed analysis of the impact of latency on the performance of Big Data applications with respect to different I/O phases (i.e., input, intermediate and output data) of an application. Although these studies mention contention as a problem, none of them investigate its impact on the application performance. Hence, we aim to complement those studies by providing a detailed analysis of the impact of latency and contention on the performance of Big Data applications. Furthermore, we show potential performance issues specific to different PVFS configurations.

3.5 Conclusions

We have recently witnessed an increasing trend towards leveraging HPC systems for Big Data processing. In this chapter, we undertook an effort to provide a detailed analysis of performance characteristics of Big Data applications on HPC systems, as first steps towards efficient Big Data processing on HPC systems. Our findings demonstrate that one should carefully deal with I/O management issues regarding HPC systems (i.e., latency and contention) when running Big Data applications on these systems. An important outcome of our study is that negative impact of latency on the application performance is present for all I/O phases. We further show that contention is a limiting factor for the performance of Big Data applications. To enable a better understanding of this phenomenon, we further investigate the root causes of I/O interference on HPC systems in Chapter 4. Lastly, we reveal that enabling synchronization for PVFS in order to provide resillience can create a serious performance bottleneck for Big Data applications.

We summarize our findings in Table 3.2. We believe that these findings can help to motivate further research leveraging HPC systems for Big Data analytics by providing a clearer understanding of the Big Data application characteristics on HPC. In this direction, by considering the performance bottlenecks of Big Data applications on HPC systems that we highlighted in this chapter, we introduce a burst buffer solution (described in Chapter 5) that tries to mitigate these bottlenecks and thus to improve the performance of Big Data applications on HPC systems.

The Impact of I/O Latency
We confirm that I/O latency resulting from the remote data access to the parallel file system leads to a significant performance degradation for all the Big Data workloads. However, in contrary to existing studies [25, 73], we demonstrate that intermediate data storage is not the major contributor to this latency problem. We also observe that the impact of this latency problem depends on the characteristics of the Big Data applications (e.g., map-heavy, iterative applications).
The Role of Contention
We demonstrate that contention appears as a limiting factor for Big Data applications on HPC systems due to employing a shared storage system. In Chapter 4, we further investigate the causes of this problem and try to understand its implications on I/O management solutions for performing efficient Big Data processing on HPC systems.
The Impact of the File System Configuration
Parallel file systems are equipped with several features which are important for HPC applications (i.e., synchronization feature in PVFS to provide resiliency, distributed locking mechanism in Lustre to ensure file consistency). However, as we demonstrated in our experiments and also reported earlier in [25, 139], these features may bring a significant performance degradation for Big Data applications. We further investigate the role of these file system specific configurations on the I/O interference in Chapter 4.

Table 3.2: Our major findings on the characteristics of Big Data applications on HPC systems.

Chapter 4

Zoom on the Root Causes of I/O Interference in HPC Systems

Contents

4.1 I/O Interference As a Major Performance Bottleneck 38
4.2 Experimental Insight Into the Root Causes of I/O Interference in HPC Storage Systems 39
4.2.1 Methodology 39
4.2.2 Experimental Results 41
4.3 Unexpected Behaviors: A Flow-Control Issue 51
4.3.1 The Incast Issue 51
4.3.2 From Incast to Unfairness 52
4.3.3 Counter-intuitive Results From a Flow-Control Perspective 54
4.4 Related Work 55
4.5 Conclusions 56

As we move closer to exascale systems, I/O interference becomes an increasingly important issue since larger machines are shared by more concurrent applications [37, 89]. In the previous chapter, we observed that I/O interference can lead to a serious performance degradation for Big Data applications. Thus, understanding the causes of I/O interference is critical to achieve efficient Big Data processing on HPC systems.

In the context of I/O for HPC systems, the main shared resource that applications contend for is the parallel file system, which comprises several components that can all become a point of contention. In this chapter, we explore these root causes of I/O interference in HPC storage systems. To this end, we conduct an extensive experimental campaign. We use microbenchmarks on the Grid'5000 [19] testbed to evaluate how the applications' access

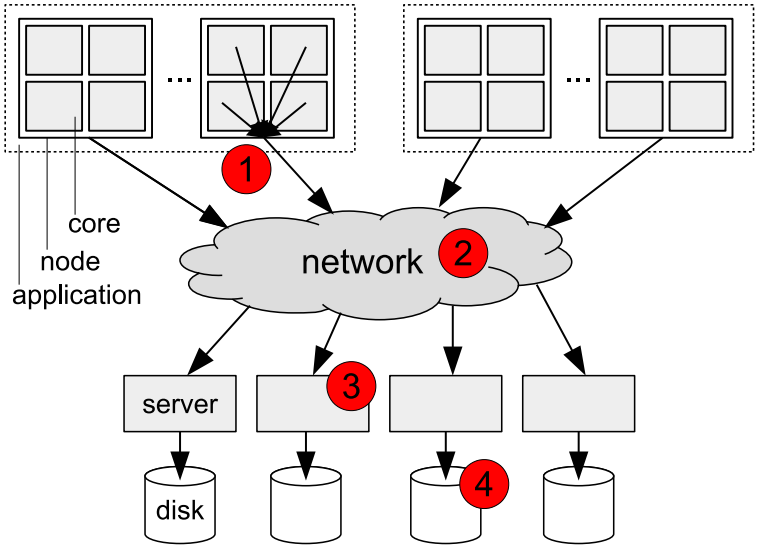


Figure 4.1: Typical parallel storage system and potential points of contention.

pattern, the network components, the file system's configuration, and the backend storage devices influence interference.

4.1 I/O Interference As a Major Performance Bottleneck

I/O interference in high-performance computing (HPC) is defined as the performance degradation observed by any single application performing I/O in contention with other applications running on the same platform. For several years researchers have been tackling cross-application I/O interference in the HPC area. The focus has been on several causes including access locality in disks [154], synchronization across storage servers [121, 154], or network contention [13, 83, 106, 125]. While these solutions get us undeniably closer to solving the I/O interference problem, they all focus on a single potential root cause of interference (e.g., the network) and do not look at the interplay between several potential causes. This section describes the potential causes of I/O interference in detail.

Points of Contention in the I/O Path

By considering the design of common HPC storage systems, we identified four potential points of contention in the data path, illustrated in Figure 4.1.

1. As the number of cores per node increases, the network interface shared by all the cores in a node can become a first point of contention [35, 89]. Failing to address this first bottleneck at a single-application level can affect the interference encountered with other applications farther down the I/O path.

2. The network linking computation nodes to storage servers (which we will call "storage network," as opposed to the "computation network" used across computation nodes,

and which may be different) is usually the first point of contention between multiple, independent applications.

3. The servers running the parallel file system constitute a third possible cause of cross-application interference, because of the limited bandwidth each server provides and because of scheduling decisions being made at this level regarding the order in which I/O requests should be served [106, 121, 154].

4. The disks (or any other backend storage devices) constitute the lowest level at which contention can occur. While the servers serialize requests into actual disk accesses, interleaved requests from different applications can break the locality of disk accesses and degrade performance [153].

Other possible sources of interference include I/O forwarding nodes [5, 68, 136], RAID technology used in backend storage devices, or the computation network.

4.2 Experimental Insight Into the Root Causes of I/O Interference in HPC Storage Systems

Investigating the root causes of I/O interference is a challenging task given the complexity of HPC storage systems and the large number of parameters that can contribute to interference. In the previous section, we identified the potential points of contention in the data path. Here, we first describe our methodology for investigating these parameters in order to draw meaningful and useful conclusions from our experiments. Then, we present the results of our investigation.

4.2.1 Methodology

Role of Each Point of Contention

One way of studying the effect of potential points of contention consists of carefully isolating each of them and benchmarking them separately. This is not the approach we undertake here. This approach indeed does not capture the interplay between causes, such as the fact that contention at one level can either mitigate or exacerbate interference at another level.

Our approach consists of either ruling out potential causes of interference or modifying their parameters and observing the resulting performance under congestion. This approach has proved much more useful not only in understanding the role of each point of contention but also in evaluating their interactions. More specifically we proceed as follows for each level.

1. The network interface can be ruled out by making a single core on each node issue all the I/O requests of that node.

2. While the network can be ruled out by having clients run on the same node along with a single-server file system, this option gives us little information about the role of the network in a large, multiserver deployment of the file system. We therefore evaluate the impact of the network's bandwidth on the interference as well.

3. The servers can be ruled out by ensuring that each group of processes accesses a distinct set of servers. In this scenario the two groups will interfere at the network level but not in the servers or for the access to the disks.

4. The disks can be ruled out by using much faster devices such as SSDs or local memory or by asking the file system to throw away any incoming data instead of storing it. Another option is to turn off the synchronization of data files in the file system, which allows the servers to keep data cached in memory and flush them to disks later. In the previous chapter, we observed that synchronization feature can significantly degrade the performance of Big Data applications therefore we further investigate its role on the I/O interference.

Microbenchmark and Reporting Method

To investigate the influence of various parameters on the I/O interference, we follow the methodology used in [37]. We developed a microbenchmark similar to IOR [119]. This application starts by splitting `MPI_COMM_WORLD` into two groups of processes running on two separate sets of nodes. Each group of processes executes a series of collective I/O operations following a specified pattern, simulating two applications accessing the file system in contention. We measure the time taken by each group of processes to complete its set of I/O operations.

The experiments presented in this chapter focus on write/write interference only as write operations typically exhibit the worst I/O performance. They use two different access patterns.

Contiguous. In this pattern, each process issues a 64 MB write request in a contiguous way in a shared file, at an offset given by $rank \times 64MB$. MESSKIT and NWChem applications, which calculate the electron density around a molecule, are some of the scientific applications that follow a contiguous pattern in their I/O accesses [120].

Strided. We represent the noncontiguous case by a one-dimensional strided access pattern. Each process issues 256 requests of size 256 KB each. This I/O pattern can be observed in scientific applications such as ESCAT and quantum chemical reaction dynamics (QCRD) application [120].

Our experimental evaluation leverages the concept of Δ-graphs introduced in [37]. For a given configuration of the platform and the microbenchmark, we introduce a delay Δ between the beginning of the I/O burst of the first group of processes and the beginning of the I/O burst of the second group. We then plot the time to complete an I/O phase as a function of Δ.

We note that Δ-graphs do not represent timelines; each point in a Δ-graph represents a single experiment.

Platform Description

The experiments were carried out on the Grid'5000 [19] testbed. We used the Rennes site; more specifically we employed nodes belonging to the *parasilo* and *paravance* clusters. The

Table 4.1: Time taken by an application running on one core to write 2 GB locally using a contiguous pattern, alone and in the presence of another application performing the same access to another file at the same moment.

Device	Alone	Interfering	Slowdown
HDD, sync ON	13.4 s	33.4 s	2.49×
SSD	2.27 s	4.46 s	1.96×
RAM	1.32 s	2.09 s	1.58×

nodes in these clusters are outfitted with two 8-core Intel Xeon 2.4 GHz CPUs and 128 GB of RAM. We leverage the 10 Gbps Ethernet network that connects all nodes of these two clusters. Reserving these two clusters and deploying our own file system ensured that we were the only users of the network switch as well as the file system at the time of the experiments.

The OrangeFS file system (a branch of PVFS2 [110]) version 2.8.3 was deployed on 12 nodes of the *parasilo* cluster. We considered two types of configuration: "Sync ON" and "Sync OFF", which represent whether each request is immediately flushed to the backend storage devices or whether data can stay in kernel-provided buffers, respectively.

We use 60 nodes (960 cores) to run our microbenchmark on the paravance cluster, unless otherwise specified. These cores will always be split into two groups of equal size (30 nodes) and follow the same type of access pattern.

4.2.2 Experimental Results

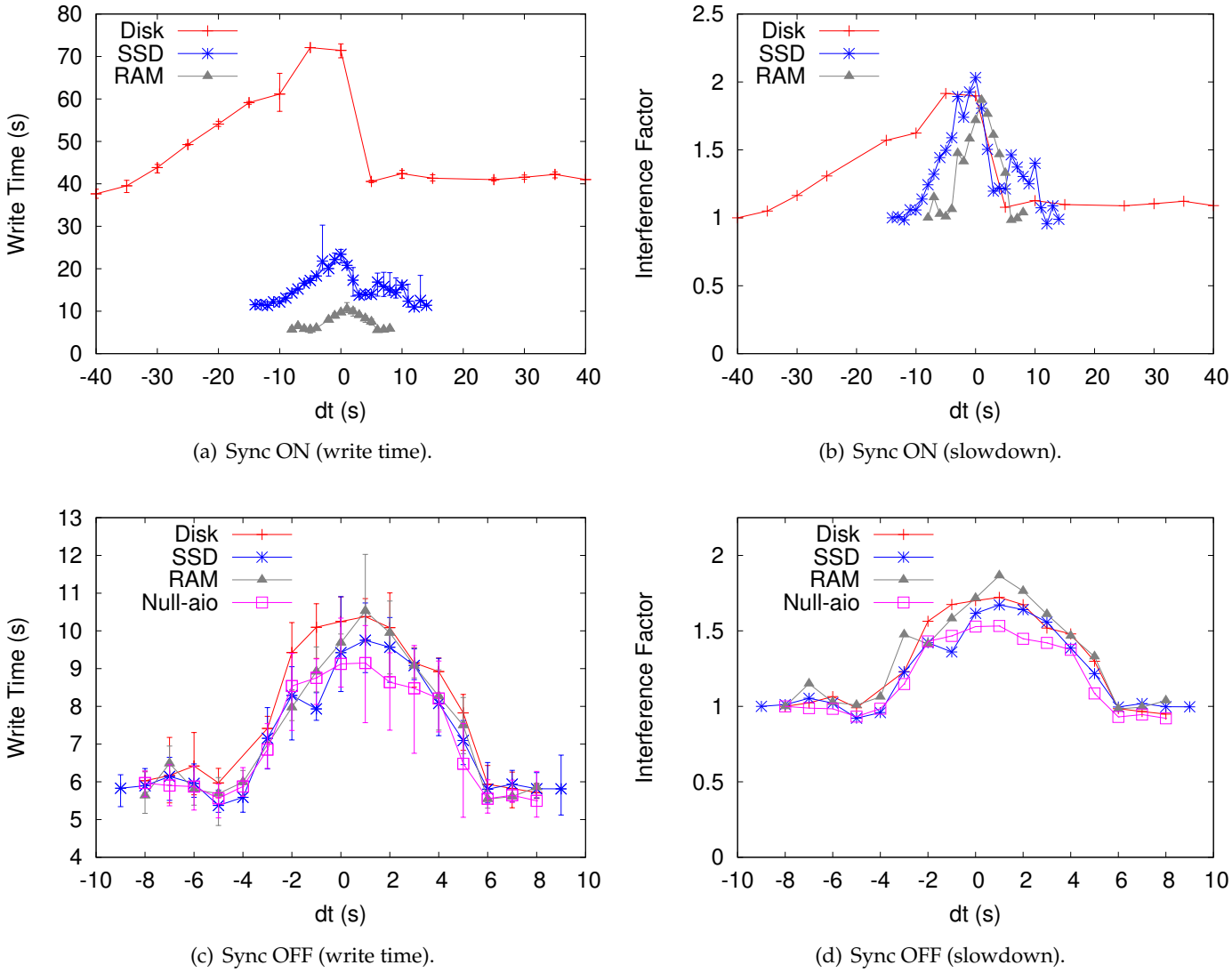
This section explores the role that each of the components presented above has in the I/O interference. We discuss several possible causes for I/O interference.

Influence of the Backend Storage Device

To investigate the I/O interference caused by the storage backend (i.e., disk-level interference), we ran our microbenchmark on the same node as the file system, with the file system deployed on this node only. This removes the network from the factors contributing to the interference and highlights disk-level interference. Each application consists of a single client writing 2 GB contiguously in a file (one file for each client). Table 4.1 shows the resulting write time and slowdown for different storage backends: HDD, SSD, and RAM.

Finding: We confirm that the use of hard disks leads to an important relative performance degradation in the presence of contention. This interference, less present when using SSDs or local memory, may stem from the additional disk-head movements produced by interleaved requests to distinct data files. Hence, use of these high throughput storage devices can help to enable performing efficient Big Data processing on HPC systems, as we explore in the following chapter.

Figures 4.2 and 4.3 complement our study of interference at the level of backend storage devices, this time with real parallel applications and file system. In both figures, two applications of the same size (480 cores each) write 64 MB per process, in a contiguous pattern in Figure 4.2 and in a strided pattern in Figure 4.3.



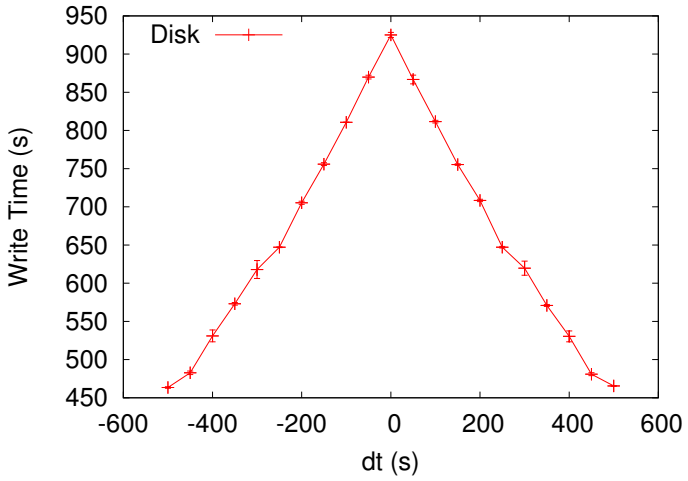
(a) Sync ON (write time).

(b) Sync ON (slowdown).

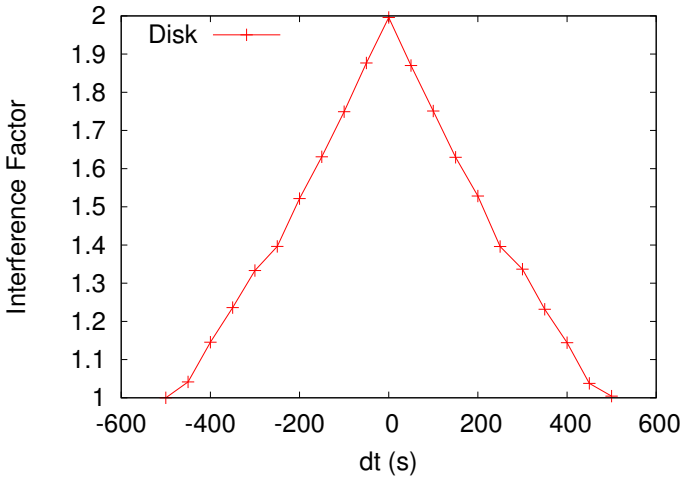
(c) Sync OFF (write time).

(d) Sync OFF (slowdown).

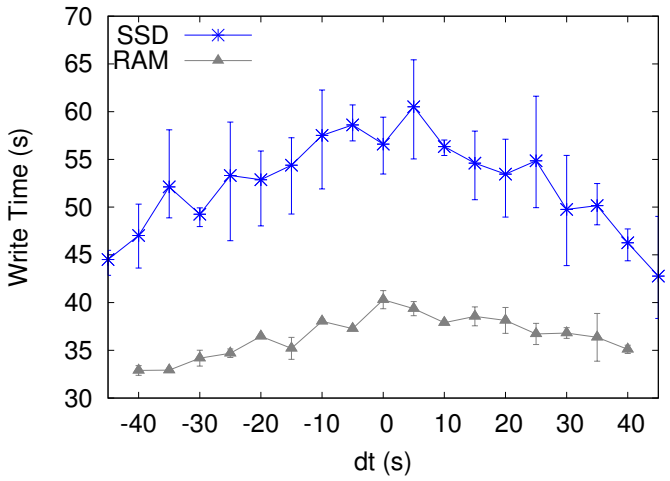
Figure 4.2: Two applications of the same size (480 cores each) write 64 MB per process using a contiguous pattern. We show how the application behaves for the different storage characteristics: disk, SSD, and RAM. Sync is enabled in (a) and (b) and disabled in (c) and (d). (c) and (d) also display the null-aio method for performing I/O, which simply does no disk I/O at all.



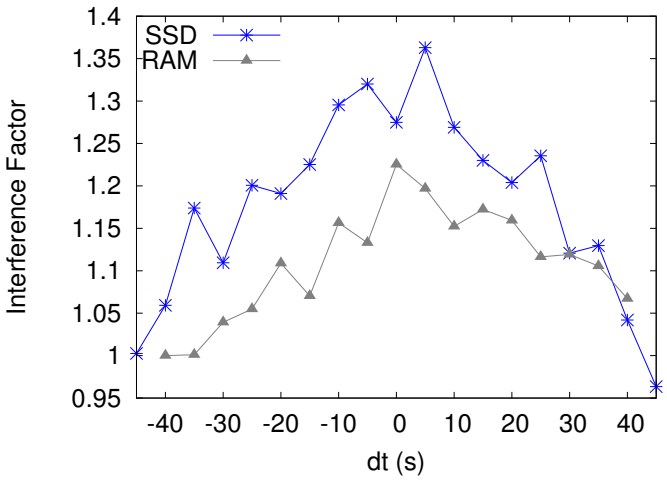
(a) HDD, Sync ON (write time).



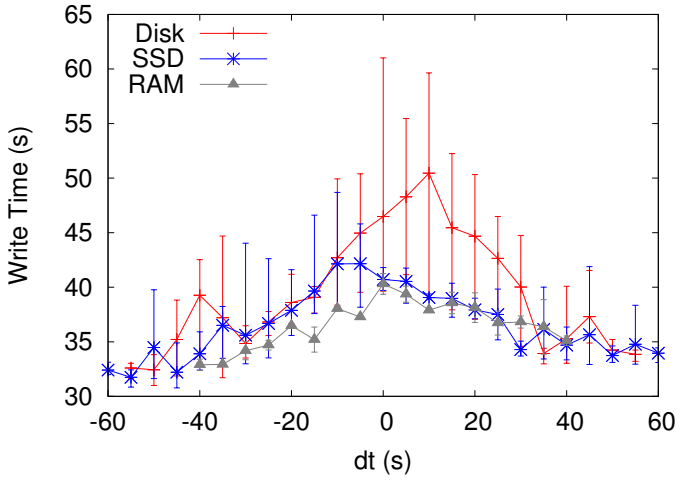
(b) HDD, Sync ON (slowdown).



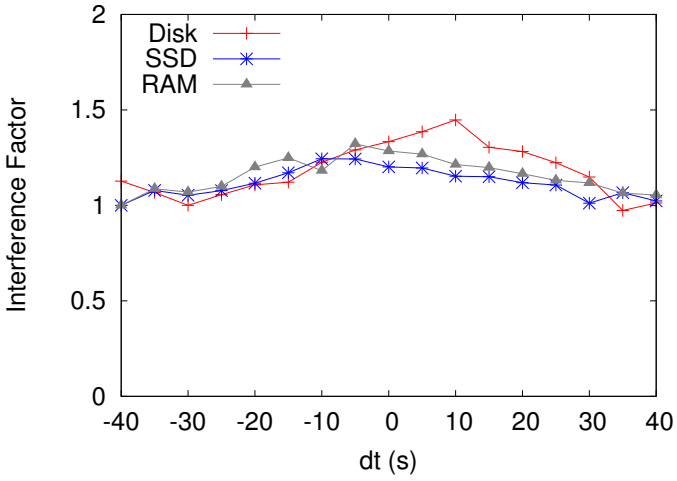
(c) Others, Sync ON (write time).



(d) Others, Sync ON (slowdown).



(e) Sync OFF (write time).



(f) Sync OFF (slowdown).

Figure 4.3: Two applications of the same size (480 cores each) write 64 MB of data per process to PVFS using a strided pattern. Due to the large write time when using hard disks and synchronization is enabled, we separated the figures for HDDs and other devices. For brevity, only 1 application is shown since both applications have the same size and behavior).

As we expect, local memory and SSDs perform better than hard disks. We also confirm our observations in the previous chapter that enabling synchronization leads to a significant performance degradation. In terms of interference, the slowdown is equivalent (up to 2×) regardless of the storage backend for a contiguous pattern.

The contiguous scenario with synchronization enabled (Figures 4.2(a) and 4.2(b)) shows an interesting result: when using HDDs (and to some extent SSDs), the graph becomes asymmetrical. That is, the first application entering an I/O phase gets better performance than does the second, although their I/O patterns are the same. Such unfair interference behavior will appear again in other scenarios throughout this section and will be further explained in Section 4.3.

Experiments with a strided access pattern and sync enabled show that local memory and SSDs have a lower interference factor compared with that of HDDs. This different behavior stems from the greater tolerance of local memory and SSDs to random accesses produced not only by interleaved requests from distinct applications but also by the strided patterns of the applications themselves.

When the synchronization is disabled, we do not observe any significant difference in terms of performance and interference for both access patterns. This is expected since the amount of the generated data is small enough to stay in the local memory when the synchronization is disabled.

Finding: Depending on the type of storage device, the access pattern may have an influence on the I/O interference behavior. While the peak interference factor is almost equal for all storage devices with a contiguous pattern, a strided pattern leads to higher interference in HDDs.

Given the regularity and symmetry of the Δ-graph with HDD and synchronization enabled (perfectly triangular figure with an exactly 2× slowdown when both applications start at the same time), we hypothesize that while in these conditions the hard disks are the points of contention, other backends are fast enough to deal with the congestion, and the slowdown observed in these situations comes from another component, such as the network. This could explain the asymmetry in those cases. This hypothesis will be examined in the following two sections.

Influence of the Network Interface

The network interface in increasingly multicore nodes is already a limiting factor to single-application I/O performance. We explored its role in cross-application interference. Figure 4.4 illustrates two scenarios: one in which all cores write 64 MB and one in which one core per node performs an equivalent amount of I/O (16× 64 MB).

We note that, as expected, the performance without interference is improved by using a single core per node instead of all the cores. This result is in line with the results of our related work focusing on dedicated I/O cores [35].

In terms of interference, having all the cores perform I/O not only produces more interference but also leads to unfairness. Indeed the interference pattern observed in Figure 4.4 with 16 writers per node is asymmetrical, which shows that the first application entering an I/O burst performs better than the one that follows it.

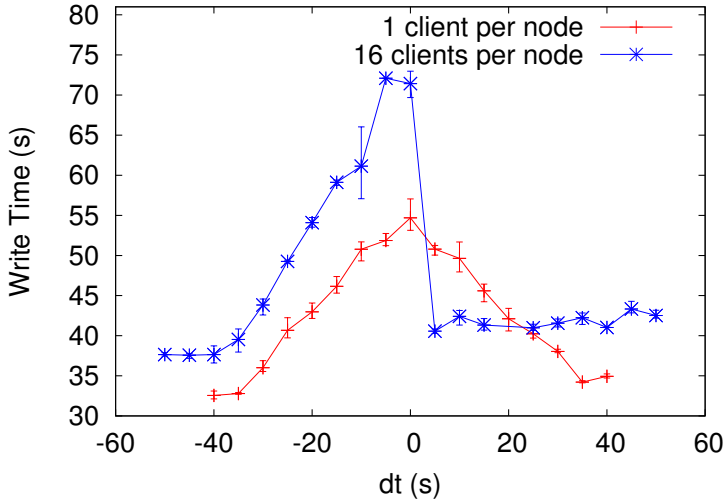
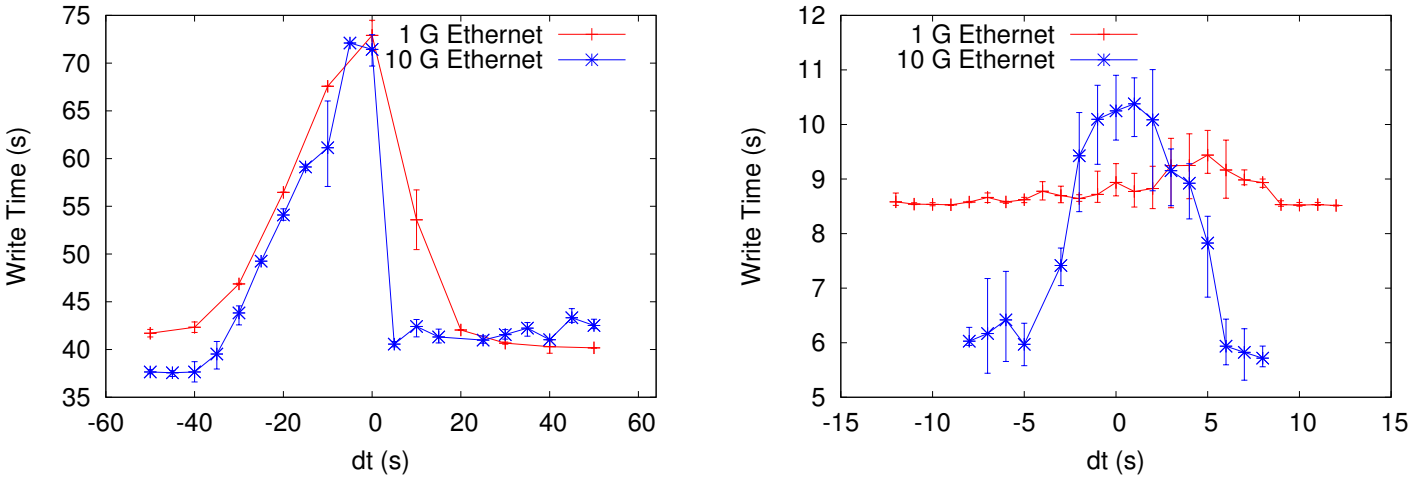


Figure 4.4: Two applications of the same size (480 cores) write data to PVFS using a contiguous pattern. We show the $\Delta-$graph when all cores participate by writing 64 MB blocks (16 clients per node) and when a single client per node writes 16 blocks of 64 MB.



(a) Sync ON.

(b) Sync OFF.

Figure 4.5: Two applications of the same size (480 cores each) write 64 MB per process using a contiguous pattern. Sync is enabled in (a) and disabled in the (b). We show the $\Delta-$graph when the network bandwidth is 10 G (default) and adjusted to the 1 G Ethernet.

Finding: While it was already known that fewer writers per multicore nodes (e.g., aggregators or dedicated I/O processes) improve the I/O performance of a single application, we have shown here that this approach also lowers cross-application interference.

Influence of the Network

We can hardly rule out the network from the HPC system because it provides the link between the computation and the storage nodes. Hence, we highlighted its role by decreasing the network bandwidth from 10 G to 1 G. Figure 4.5 shows the results for the different network bandwidths, with two applications writing in a contiguous pattern. Surprisingly, we discover that having a higher network bandwidth neither significantly improves the applications' I/O performance nor helps eliminate the interference. On the contrary, limiting the network bandwidth to 1 G helped eliminate the interference when synchronization was disabled in the disks, as shown in Figure 4.5(b), and helped regain a symmetrical (fair) interference behavior when synchronization was enabled, as shown in Figure 4.5(a).

In Figure 4.5(a), the peak write time in the presence of contention is the same whether we use a 10 G or a 1 G network. The reason is that the performance of the I/O path here is limited by the backend storage devices (HDDs). In Figure 4.5(b), the data is not synchronized to disks right away when reaching storage servers but stays in buffers. Hence, the network becomes the limiting factor.

The flat Δ-graph observed with a 1 G network stems from the fact that the network is limiting the rate at which each application sends requests to the file system, producing an interference-free behavior. Interesting is the fact that the resulting write time is in some cases smaller than when using a 10 G network, which hints for enabling efficient Big Data processing on HPC systems that constraining the rate at these applications send their data to the parallel file system can be a valid solution for mitigating interference.

The fairness regained in Figure 4.5(a), resulting from the interplay between the storage devices and the network, will be further explained in Section 4.3.

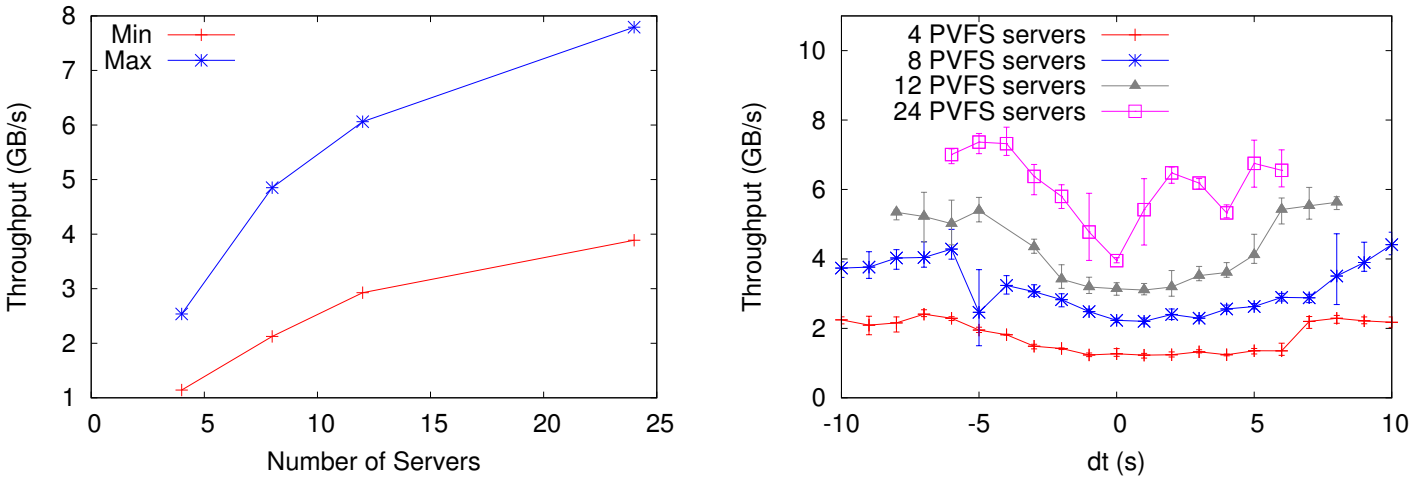
Finding: Counterintuitively, a lower network bandwidth may not cause higher interference. On the contrary, it can prevent interference if none of the other components are subject to contention.

Influence of the Number of Storage Servers

Intuitively and assuming the network is not a point of contention, using more storage servers increases the aggregate throughput that any single application can achieve. This situation is demonstrated by the maximum throughput achieved in Figure 4.6(a). In terms of interference, however, it is not clear whether more servers will mitigate the interference.

We therefore investigated the role of the number of servers on the interference by deploying PVFS on 24, 12, 8, and 4 nodes with synchronization turned off. Each client writes 64 MB in a contiguous pattern for the first three deployments and writes 32 MB with 4 PVFS servers because of its lower capacity. Figure 4.6(b) shows the observed throughput for one of the applications depending on the number of PVFS servers used and on Δ. As expected, increasing the number of servers improves the throughput, but it cannot eliminate the interference. I/O interference still exists because each server still has the same number of clients regardless of the number of servers.

Table 4.2 summarizes the peak interference factors observed for each number of servers. As we can see, the number of servers does not influence the interference factor much.



(a) Scaling.

(b) Δ-graph.

Figure 4.6: Two applications of the same size write data to the PVFS using a contiguous pattern. Figure (a) shows the maximum throughput achieved (when the application is alone) and the minimum (in contention) as a function of the number of servers. Figure (b) shows the throughput for one of the applications in a Δ-graph, that is, as a function of the delay between applications.

Table 4.2: Peak interference factor observed by the application for different numbers of storage servers.

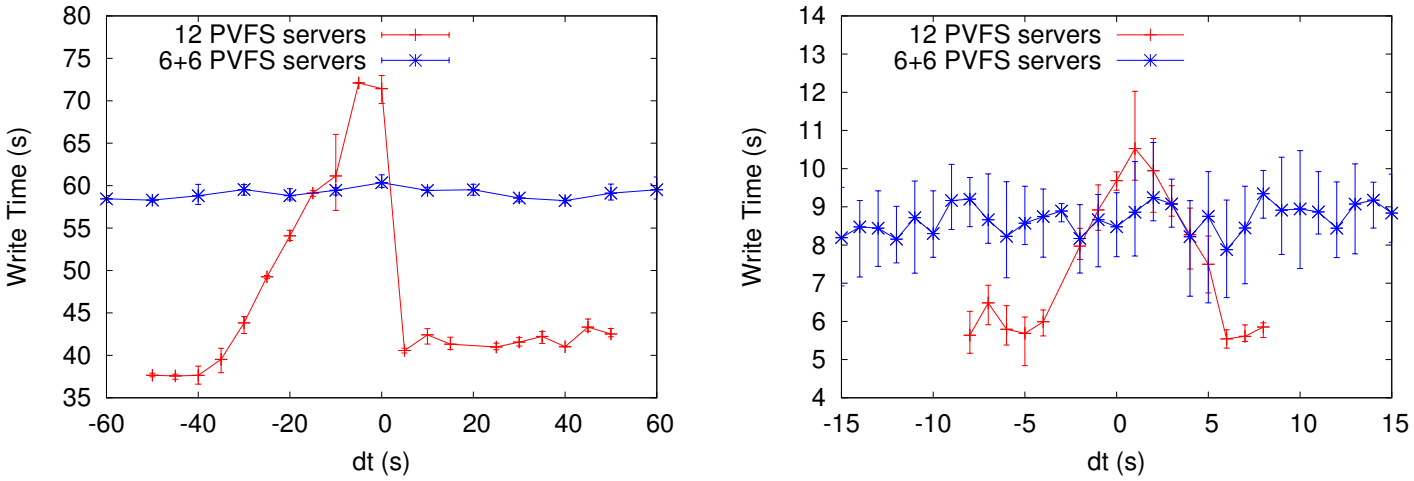
Number of Servers	Interference Factor
4	2.22
8	2.28
12	2.07
24	2.00

Finding: Increasing the number of servers does not affect the relative performance degradation generated by cross-application interference.

One could argue that this result would not be true for small applications that cannot get full parallelism from the maximum number of storage servers. Yet as we build larger machines, the number of storage servers tends to get smaller relative to the number of computation nodes. Moreover, with the Big Data deluge, we tend to run larger applications that become quickly limited by this small number of servers.

Influence of Targeted Storage Servers

In our previous set of experiments, both applications were writing to all available servers. In this section, we split the 12 PVFS servers into two groups so that each application targets a different group of servers. Our idea is to remove the servers and disks from the possible points of contention, leaving only the network as a shared component between the two



(a) HDD, sync ON.

(b) RAM.

Figure 4.7: Two applications of the same size (480 cores each) write 64 MB per process using a contiguous pattern. We show the $\Delta-$graph when both applications use the same set of PVFS servers (12 PVFS servers) and when each application targets different set of PVFS servers (6+6 PVFS servers) for the two different storage backends.

applications.

Figure 4.7 shows the results for two different storage backends. As expected, using $2\times$ fewer servers decreases the performance of a single application.

The behavior with respect to the interference is more interesting, however. We observe that making each application target a different set of servers removes the interference. In some cases, the interference observed under contention for all 12 servers leads to a higher write time than does using 6 separate servers for each application. This result would motivate approaches that detect potential congestion and partition the storage space across applications instead of letting applications interfere.

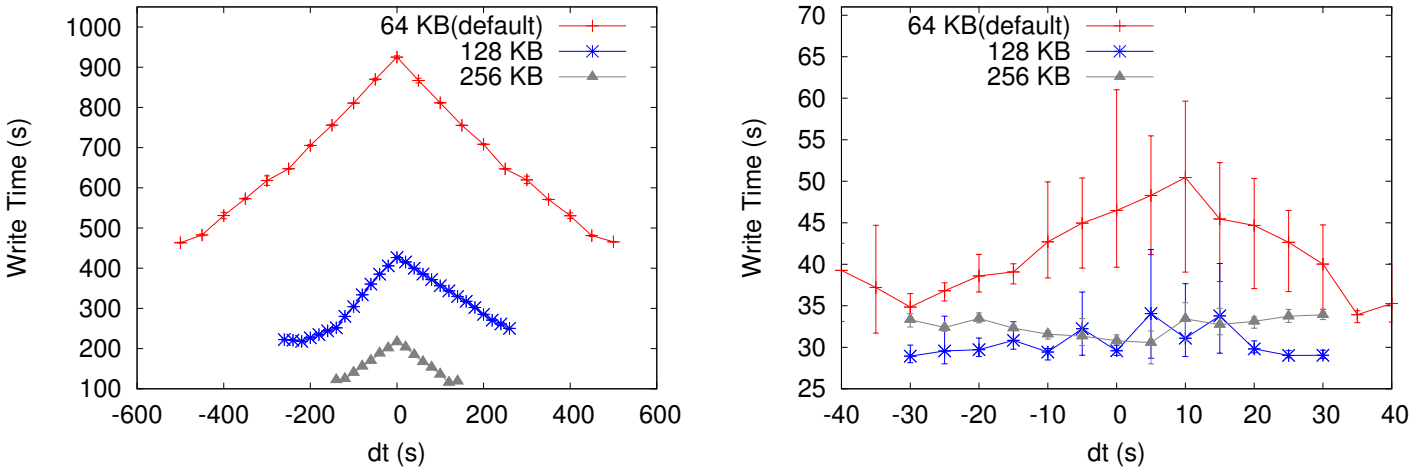
Again, the unfairness observed in Figure 4.7(a) when both applications target all servers is eliminated when they access different sets of servers.

Finding: Making distinct applications target distinct sets of servers is a valid solution to at least control if not mitigate the level of interference.

Influence of the Data Distribution Policy

Data files are distributed across PVFS servers in a round-robin fashion with a predefined stripe size. Changing this stripe size can have a significant impact on the resulting performance. Hence we wanted to check its influence on the interference as well.

Figure 4.8 illustrates the interference pattern with stripe sizes of 64 KB, 128 KB, and 256 KB for a PVFS deployment with disk and synchronization enabled (Figure 4.8(a)) and disabled (Figure 4.8(b)). Note that 64 KB is the default stripe size and that each application writes 64 MB per process in a strided pattern with 256 KB of blocks.



(a) Sync ON.

(b) Sync OFF.

Figure 4.8: Two applications of the same size (480 cores each) write 64 MB of data to the PVFS using a strided pattern, with different stripe sizes on the server side. Synchronization is enabled in (a), and disabled in (b).

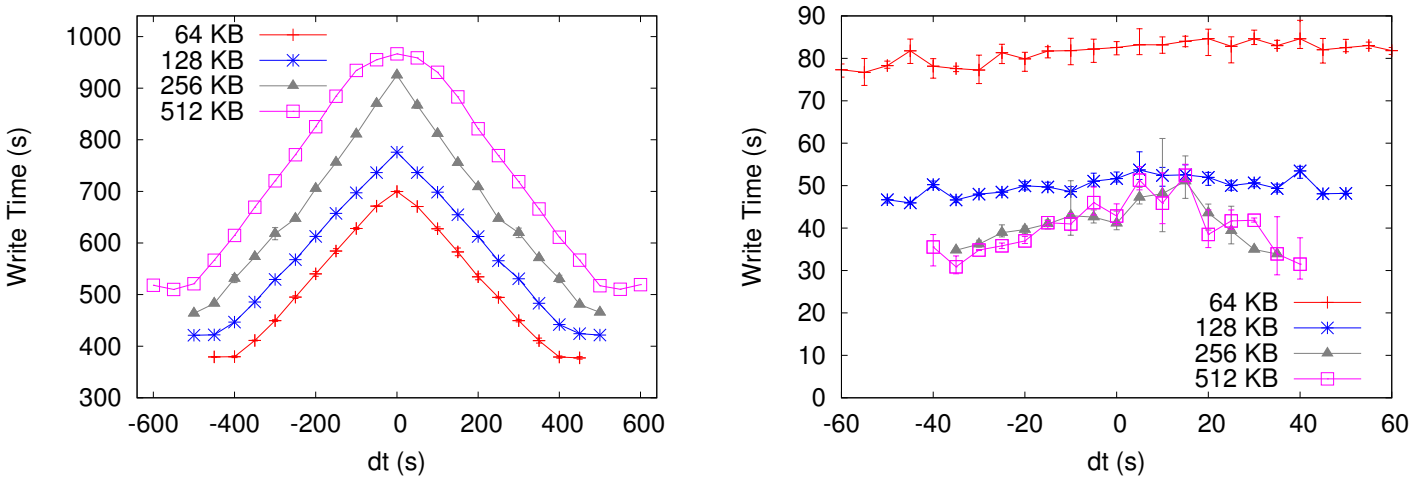
A stripe sizes higher than the default one leads to significant performance improvements for both cases. However, the interference seems to disappear when using a larger stripe size with synchronization turned off. We hypothesize that this lower interference stems from the smaller number of servers that each request is striped across. When a large request is issued by a client, this request is split into smaller requests sent in parallel to several servers. The operation completes only when all these servers have treated their part of the initial request. Hence, any slowdown experienced by a single server as a result of contention leads to a global slowdown for the entire operation. Provided that two servers decide to serve requests from different applications in a different order, both applications will suffer from a slowdown observed in servers that have not prioritized their request.

Here each 256 KB request is striped across 4 servers for the default stripe size of 64 KB. This is reduced to 2 servers with a 128 KB stripe size and to 1 with a 256 KB stripe size. Hence, we see the performance improvement for both cases and the removal of I/O interference for the disabled sync case. We believe that interference still exists for the other scenario since disk is still an active component and contributing to the I/O interference.

Finding: We confirm that making all servers treat requests from distinct applications in the same order, as done in [121], is an appropriate way of mitigating I/O interference.

Influence of the Request Size

Similarly to the stripe size in the file system, the original request size in applications has an impact on I/O performance. Figure 4.9 illustrates the interference patterns when each application writes 64 MB in a strided pattern with a block sizes of 64 KB, 128 KB, 256 KB, and 512 KB. The stripe size in PVFS is set to the default of 64 KB.



(a) Sync ON.

(b) Sync OFF.

Figure 4.9: Two applications of the same size (480 cores each) write 64 MB of data to the PVFS using a strided pattern with a block sizes of 64, 128, 256 and 512 KB. Synchronization is enabled in (a) and disabled in (b). These graphs show the write time for the application (for brevity, only 1 application is shown since both applications have the same size) depending on the block size used and on dt.

The best performance is achieved for small block sizes when synchronization is enabled (Figure 4.9(a)), whereas it is achieved with large block sizes when synchronization is disabled (Figure 4.9(b)).

While the interference pattern shows a fair, proportional sharing of resources for all block sizes when synchronization is enabled (symmetric, triangular figure), when synchronization is disabled the interference pattern disappears for block sizes of 64 and 128 KB. This result is in line with our observations made in Section 4.2.2 and the fact that such request sizes have fewer servers involved in each I/O operations. Yet while these small request sizes remove the interference, they are far from optimal for a single application.

Finding: The fact that no interference is observed between two applications does not mean that optimal performance is achieved. Our experiments show that, while some request sizes allow cross-application interference to be mitigated because clients interact with fewer servers for each request, these block sizes remain far from optimal from a single-application perspective.

Following this observation, we warn any researcher proposing solutions to the I/O interference that these solutions should be validated in configurations that are already as good as possible, if not optimal, for a single application alone. Indeed one can claim that a solution removes the interference, while much higher performance could actually be obtained from each application individually by better optimizing their access patterns.

4.3 Unexpected Behaviors: A Flow-Control Issue

This section explains further some of the results, in particular the counter-intuitive ones, and highlights a flow-control issue at the core of the interplay between components.

4.3.1 The Incast Issue

Some of the results of the previous section remain unexplained, such as the unfairness of some scenarios, with the application that starts first getting better performance than the one starting second. An unfair behavior results from a component that adapts to the workload *over time*. Since PVFS does not implement any particular scheduling mechanism at the server side, there is no reason to think one flow of requests from an application would be prioritized over another. The prioritization of one flow over another cannot result from the backend storage device either, since this storage device sees only serial accesses from a single program: the PVFS server. Hence we suspected that such unfair behaviors stem from the network.

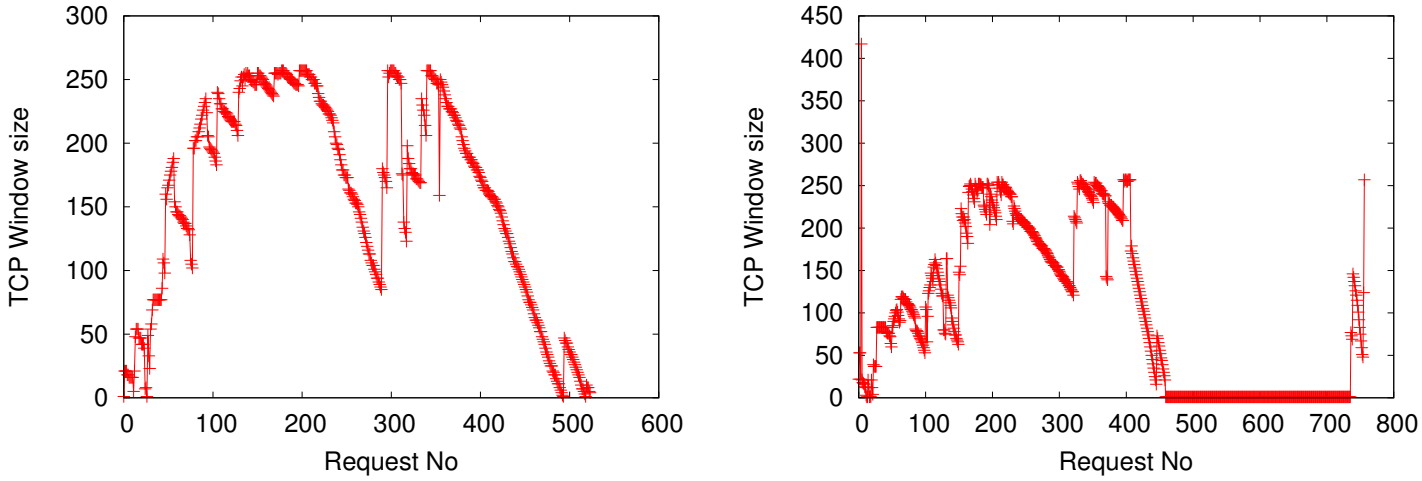
To confirm this hypothesis, we reran the experiment presented in Figure 4.2(a), in which two applications of 480 cores each write 64 MB per process in a contiguous manner, in a PVFS file system consisting of 12 servers. We examined more closely the TCP packets exchanged between a client of either application and a PVFS server, using *tcpdump*.

Figure 4.10(a) shows the evolution of the TCP window size for the sequence of requests issued by one client to one server, when an application runs alone. Figure 4.10(b) shows the evolution of the TCP window size when the application is interfering with another one. As we can see, the behavior is similar except for the fact that, under contention, the window size drops to nearly 0, making it difficult for the client to eventually send all its data.

The collapse of the TCP window size as a result of contention was shown by Phanishayee et al. [102], who termed it the "Incast problem." When many clients access to a server, the TCP congestion control mechanism at this server forces the window size to drop in all its opened sockets, leading to an important loss of performance.

Note that this phenomenon does not stem from the network alone (we have seen, by splitting the set of servers into two groups, that the network is not a point of contention). It comes from the interplay between the network and the disks, as well as the lack of flow control mechanism in Trove, the component of PVFS that forwards requests from sockets down to the storage devices. Because disks are slow, Trove cannot keep up with the flow of incoming requests and hence relies on the TCP congestion control mechanism to limit the flow the requests from all clients.

Finding: The fact that the Incast problem appeared only with HDD and synchronization enabled, but not with RAM or SSD, proves that this type of interference results from the interplay between several components of the I/O path. What appears to be a network congestion issue can actually stem from bad flow-control induced by slow backend storage devices.



(a) Independent run.

(b) Interfering.

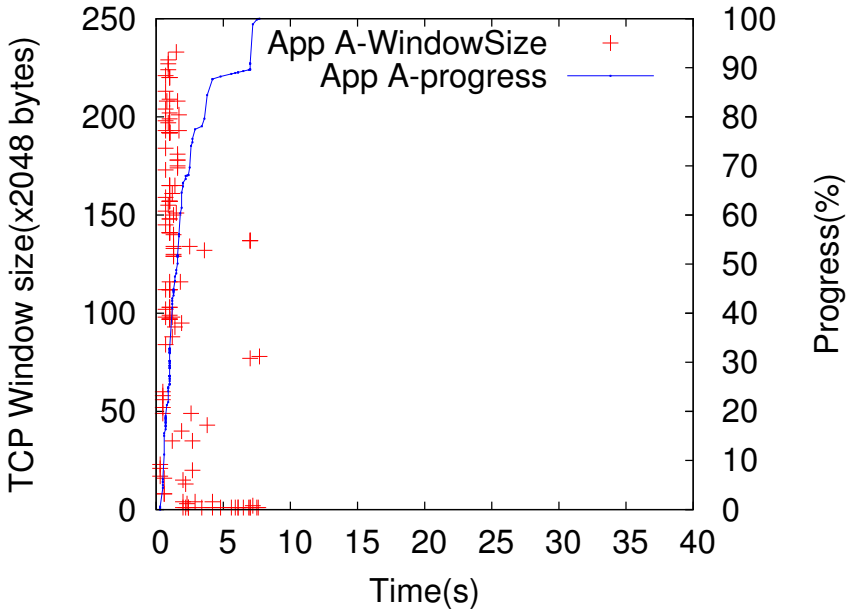
Figure 4.10: TCP window sizes of each request during the 64 MB contiguous data write from a client to the server (a) when the application is running independently and (b) interfering with the other application, which also has same size (480 cores each) and started at the same time (dt=0).

4.3.2 From Incast to Unfairness

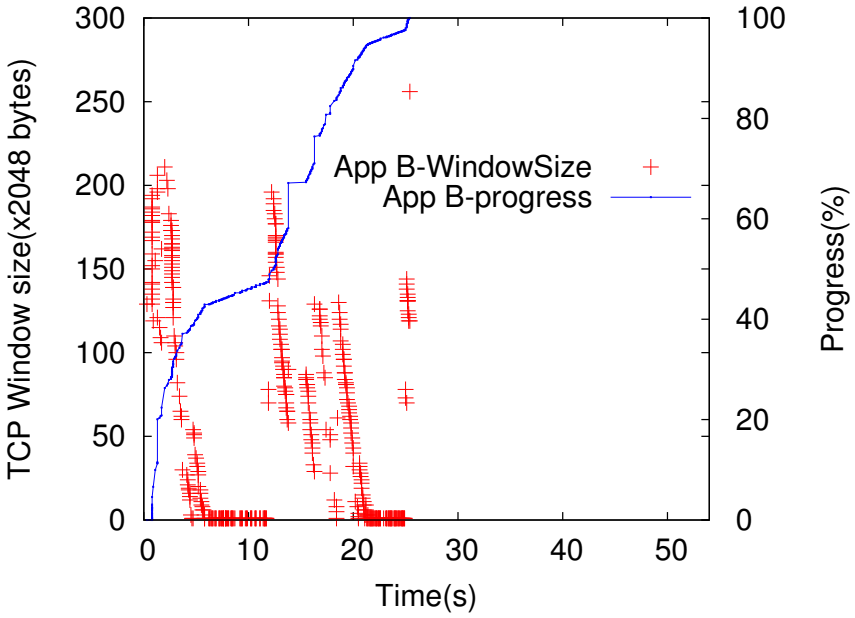
This Incast issue explains many of the results presented in the previous section, starting with the unfairness observed in some scenarios. Figure 4.11 shows the behavior of two applications, one of which starts 10 seconds after the other. We plot the TCP window size and the progress of the I/O transfer to this server as a function of time, for one client of each application. Whereas the first application starts seeing a slowdown of its progress at around 90 percent, the slowdown can be observed at 40 percent for the second application; indeed, the window size hardly manages to get back to a high value.

The appearance of unfair behavior as a result of Incast is a good way to evaluate the conditions that cause Incast to appear. For example, Figure 4.12 shows the interference behavior when running different numbers of clients. An interesting observation is that while the unfair behavior benefiting the first application is clear when using 960 or 704 clients in total, the trend seems to reverse at smaller client counts (256 to 512 clients), where the first application is more impacted than the second one.

We hypothesize the following explanation. At large client counts, the TCP window size of the second application immediately collapses as a result of contention, allowing the first application to complete seemingly without contention. At intermediate client counts, the TCP window size is reduced in both applications but does not collapse. Thus the first application is impacted as well. At small process counts, the servers are able to handle all the requests without having to shrink the window size. The interference observed becomes that of the backend storage devices.



(a) Application A



(b) Application B

Figure 4.11: TCP window sizes and progress of the data transfer from one client of each application and one of the servers.

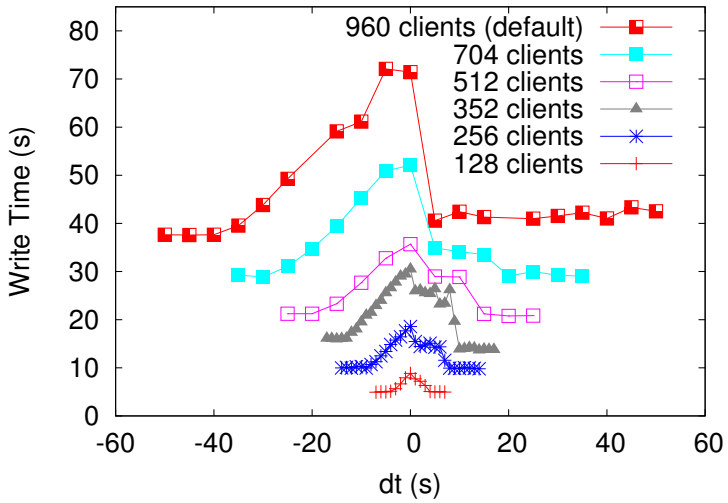


Figure 4.12: Δ-graph illustrating the appearance of the Incast problem as we increase the number of clients. Each application writes 64 MB per process in a contiguous pattern. PVFS is deployed on 12 servers with hard disks as backend, synchronization enabled. The number of clients shown is the total number of clients.

Finding: As the number of clients increases, we are more likely to observe degenerated flow-control issues as a result of the file system not being able to handle the load.

4.3.3 Counter-intuitive Results From a Flow-Control Perspective

The interplay between components resulting in a bad flow control can explain both the intuitive and counterintuitive results obtained previously.

Using one core per node instead of all cores to perform I/O, as done in Section 4.2.2, reduces the number of sockets involved in an application's I/O phase and forces a serialization of requests at the level of each node. This constrains the rate at which each single node can write and therefore prevents the Incast problem from happening.

Using a low-bandwidth network as done in Section 4.2.2 with a 1 G network instead of a 10 G, also mitigates the Incast problem by constraining the rate at which each client can send requests. By forcing a reduction of bandwidth at the source, the rate of requests becomes sustainable to the backend storage devices and thus the TCP window size does not collapse.

Splitting the servers into two groups completely prevents the interference from happening (Figure 4.7(a)) because a server has to interact with 2× fewer clients, therefore maintaining flow control on 2× fewer links.

4.4 Related Work

As we move toward the exascale era, performance variability in HPC systems remains a challenge. Cross-application I/O interference is one of the major causes of this performance variability. A large body of studies have sought to eliminate cross-application I/O interference by focusing on possible sources of this interference. For example, Zhou et al. [157] present an I/O-aware batch scheduler that addresses the interference problem at the level of batch scheduling. The batch scheduler schedules and coordinates the I/O requests on the fly by considering the system state and I/O activities. Gainaru et al. [43] show the performance degradation due to I/O congestion and propose a new scheduler that tries to eliminate this congestion by coordinating the I/O requests depending on the application past behaviors and system characteristics. Boito et al. [153] propose AGIOS, an I/O scheduling library for parallel file systems. AGIOS incorporates the applications' access pattern information into the scheduler based on the traces generated by the scheduler itself and uses this information to coordinate the I/O requests in order to prevent congestion to the file system. As we observed, however, although scheduling-level solutions can help control the level of interference, it does not always lead to improved performance at the same time.

Some works focus on finding solutions at the disk level, the lowest level that I/O interference can occur in the I/O stack. Zhang and Jiang [155] point out that frequent disk head seeks, because of the access interference on each I/O node, can seriously hurt the performance of a system. They propose a data replication scheme, InterferenceRemoval, to eliminate I/O interference. InterferenceRemoval tries to limit the number of the I/O requests served by each I/O node. Although this solution is in parallel with the Incast problem we presented in our work, we observe that it is not present for only a single source (e.g., disk) of interference.

Some research efforts consider network contention as the major contributor to the I/O interference. Bhatele et al. [16] investigated the performance variability in Cray machines and found out that the interference of multiple jobs that share the same network links is the primary factor for high performance variability. Jokanovic et al. [76] introduce the concept of quiet neighborhoods, a job allocation technique based on the job sizes. This technique helps control the fragmentation in the HPC systems and reduces the number of jobs sharing the network, with the aim of minimizing the interference.

Some works study the interference problem with a special emphasis on a single factor. Kuo et al. [82] investigated the influence of the file access pattern on the degree of interference observed. They found out that chunk size can determine the degree of interference and that the interference effect induced by various access patterns in the HPC system can slow the applications by a factor of 5. Our work is different in the targeted objective, since we try to identify all possible sources of interference under various scenarios, as well as their interplay.

Although indeed important, the aforementioned studies –by focusing only on a single potential source– do not necessarily provide a complete solution for the interference problem. In contrast, we consider the possible sources of interference together and conduct an extensive experimental study. Thus, our work can provide useful insights into the I/O interference phenomenon. Furthermore, it can help researchers tackle the interference problem across all components of the I/O system.

4.5 Conclusions

I/O interference in large-scale platforms is an important problem that can affect the efficiency of an entire machine. This problem will be even more important with exascale machines that will allow more applications (i.e., Big Data and HPC applications) run concurrently. In this chapter, we investigated the potential root causes of I/O interference. Our findings demonstrate that interference results from the interplay between several components in the I/O stack. For instance, we observe that the impact of the request size on interference varies depending on the configuration of components in the I/O path. Our findings also illustrate many counter-intuitive results. For example, we show that using a low-bandwidth network in some scenarios can eliminate the interference problem, which stems from the interplay between the different points of contention. Hence, we believe that researchers must understand the tradeoffs between several components in the I/O stack and must address the interference problem in its entirety, rather than focusing on any single component.

To the best of our knowledge, this is the first work investigating the role of each of the potential root causes of interference and their interplay. We summarize our findings together with their implications on the Big Data processing in HPC systems in Table 4.3. Based on this, in the following chapter, we propose a burst buffer solution that employs interference models for both HPC and Big Data applications in order to mitigate the I/O interference problem.

Influence of the Backend Storage Device	**Implications**
RAM and SSDs perform better than hard disks. In terms of interference, however, the slowdown is equivalent (up to 2x) regardless of the storage backend.	Use of high throughput storage devices can help to improve the performance of Big Data applications. However, interference problem still exists and should be taken into consideration.
Influence of the Network Interface	**Implications**
While it was already known that fewer writers per multicore nodes (e.g., aggregators or dedicated I/O processes) improve the I/O performance of a single application, we have shown here that this approach also lowers cross-application interference.	Aggregating the I/O requests (to the parallel file system) from large number of compute nodes into smaller set of burst buffer nodes can help to lower the I/O interference.
Influence of the Network	**Implications**
Counterintuitively, a lower network bandwidth may not cause higher interference. On the contrary, it can prevent interference if none of the other components are subject to contention.	Although high-speed networks can provide low latencies for the I/O accesses, they can on the other hand impose a more severe interference problem which we need to take into consideration when employing HPC systems for Big Data processing. For instance, one effective interference mitigation strategy would be constraining the rate at Big Data applications send their I/O requests.
Influence of the Number of Storage Servers	**Implications**
Increasing the number of servers does not affect the relative performance degradation generated by cross-application interference.	This implies that we require smart I/O interference mitigation solutions more than hardware improvements such as increasing the capacity of storage servers in order to achieve efficient Big Data processing.
Influence of the Data Distribution Policy	**Implications**
We confirm that making all servers treat requests from distinct applications in the same order is an appropriate way for mitigating I/O interference.	Setting the stripe size of a parallel file system equal to the block size of the data set for the Big Data application can accelerate the performance of Big Data applications.
Influence of the Number of Clients	**Implications**
As the number of clients increases, we are more likely to observe degenerated flow-control issues as a result of the file system not being able to handle the load.	As Big Data applications employ a large number of clients to process huge amount of data, use of burst buffers can help to improve the performance of these applications by using fewer clients for performing the I/O operations of Big Data applications.

Table 4.3: Our major findings on the root causes of I/O interference and their implications on performing efficient Big Data processing in HPC systems.

Chapter 5

Eley: Leveraging Burst Buffers for Efficient Big Data Processing in HPC Systems

Contents

5.1 Limitations of Current Burst Buffer Solutions to Big Data Processing . . . 60
5.2 The Eley Approach . . . 61
5.2.1 Overview of Eley . . . 61
5.2.2 Design Principles . . . 63
5.2.3 Performance Models for HPC and Big Data Applications . . . 63
5.2.4 Interference-Aware Prefetching . . . 65
5.3 Experimental Evaluation . . . 68
5.3.1 Methodology . . . 68
5.3.2 Real System Experiments . . . 71
5.3.3 Simulation Results . . . 73
5.3.4 Summary of the Results . . . 76
5.4 Related Work . . . 77
5.4.1 Early Adoption of Big Data Processing in HPC Systems . . . 77
5.4.2 Extending Burst Buffers for Big Data Applications in HPC Systems . 77
5.5 Conclusions . . . 77

RECENTLY, we have witnessed a dramatically increasing gap between the computation and I/O capabilities of HPC systems. For example, the Trinity supercomputer that is used currently at Los Alamos National Laboratory has brought a 28-fold im-

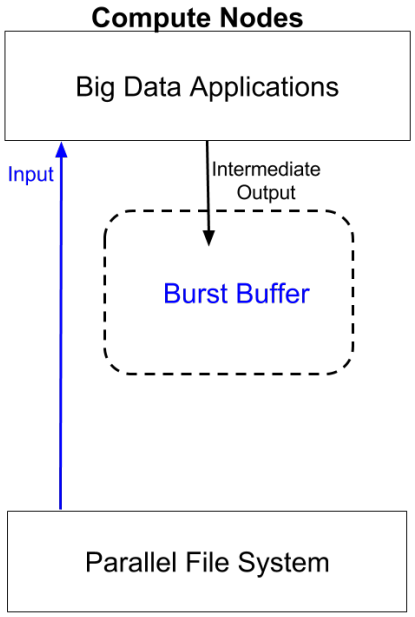


Figure 5.1: Naive adoption of burst buffers for Big Data processing in existing studies.

provement in the computation capabilities with only a 9-fold increase in the I/O capabilities compared to its predecessor, Cielo [130].

Given the widening gap in the computation and I/O capabilities of HPC systems, an important challenge in performing efficient Big Data processing on these systems is to develop effective I/O management solutions. Chapter 3 highlighted I/O latency and I/O interference as the major performance bottlenecks for Big Data applications in HPC systems. In this chapter, we propose Eley, a burst buffer solution, to alleviate these performance bottlenecks. Eley is different from current burst buffer solutions where the main focus is on the intermediate data storage and disregard the high latencies resulting from reading the input data. Moreover, these solutions do not consider the interference problem which can contribute to a significant performance degradation, not only for Big Data applications but also for the HPC applications, as we demonstrated in Chapter 4. In contrast, Eley embraces an interference-aware prefetching technique that makes reading the input data faster while controlling the interference imposed by Big Data applications on HPC applications. We evaluate Eley with both real system experiments on Grid'5000 testbed and simulations using a wide range of Big Data and HPC applications.

5.1 Limitations of Current Burst Buffer Solutions to Big Data Processing

Burst Buffer (BB) is an intermediate storage layer between the parallel file system and computation nodes, which consists of high throughput storage devices. They provide an effective solution for reducing the data transfer time and the I/O interference in HPC systems. Hence, many research efforts have been dedicated to extend BBs for Big Data appli-

cations [25, 73, 114, 139]. Figure 5.1 illustrates the overview of these works where the main focus is to overcome the high latency problem, especially for storing the intermediate data (i.e., map output for batch jobs and temporary output produced between stages for iterative jobs). For example, Islam et al. [73] utilized NVRAM as an intermediate storage layer (i.e., burst buffer) between compute nodes and Lustre which improved the application performance by 24%. Wang et al. [139] leveraged SSDs for storing the intermediate data. Chaimov et al. [25] used a separate set of nodes (burst buffer nodes) with NVRAM as a storage space to achieve a better scalability by reducing the latency when reading/writing the intermediate data. Unfortunately, the aforementioned works to extend burst buffers for Big Data applications in HPC systems may fail in practice to achieve the desired performance due to following:

- Current works ignore the latency problem in the input phase despite the significant amount of work dedicated to improve the performance of Big Data applications by sustaining high data locality when reading the input data.
- These works proposed the adoption of burst buffers in a similar way to the traditional use of burst buffers on HPC systems which aim to minimize the I/O time by absorbing the checkpointing data of scientific applications. In contrast, Big Data applications mostly run in batches therefore there is a continuous interaction with the parallel file system for reading the input data.
- In the previous chapter, we highlighted that although the use of high throughput storage devices can improve the performance of Big Data applications, interference problem still exists. Unfortunately, none of these efforts considered this interference problem which can contribute to a significant performance degradation — not only for Big Data applications but also for HPC applications — by up to 2.5x as we demonstrated in Chapter 3 and 4.

Our work addresses the limitations of current burst buffer solutions and takes a step forward toward smart I/O management of Big Data applications in HPC systems by being able to alleviate I/O latency and interference problems. To this end, this chapter presents Eley, a burst buffer solution that aims to accelerate the performance of Big Data applications while guaranteeing the QoS requirements of HPC applications. Similar to existing works that target QoS-aware execution of HPC applications [44, 95, 133], we express QoS in terms of the deadline constraint for the completion of an application. These constraints can be given by users. For instance, some applications are QoS-sensitive thus require tight deadlines for their correctness (e.g., medical imaging, video transcoding) [41].

5.2 The Eley Approach

5.2.1 Overview of Eley

With the convergence between Big Data and HPC, many Big Data applications are moving to HPC systems to benefit from their high computation capabilities. Figure 5.2 shows an architecture of a HPC system with a burst buffer enabled to address the high I/O latency challenge of Big Data applications on HPC systems. Eley is located between the compute

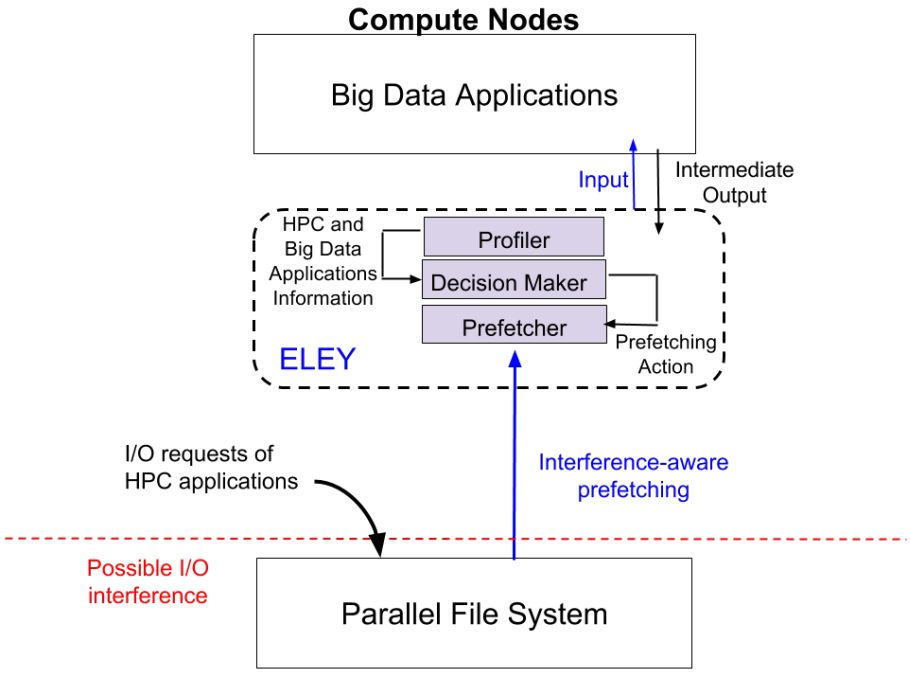


Figure 5.2: System overview of Eley.

nodes and the parallel file system where compute nodes only interact with the burst buffer nodes for all I/O phases of the Big Data application. This also includes reading the input data besides reading/writing the intermediate data, different from the existing studies on burst buffers. Thus, the interference at the file system level mainly comes from the I/O requests of burst buffer nodes and the HPC applications.

We instruct Eley to prefetch the input data of Big Data applications to optimize the I/O performance of those applications. A typical job of Big Data applications consists of several iterations (waves) depending on the total input size and the cluster capacity (i.e, the number of map tasks which can simultaneously run in the cluster). For example, if the cluster capacity is set to 100 map tasks where each map task processes one block of 128 MB, a job of 100GB data inputs will be executed in 8 waves. This is common for large scale scientific Big Data applications. For instance, the amount of processed data was almost 900 TBs and almost 100 of executed jobs have an input data of 4.9 TBs during 9 months in a research cluster (i.e., M45) with 400 nodes [52]. Thus, *Eley* employs a novel *prefetcher* component that prefetches data inputs of next iterations while computing nodes are still busy processing data inputs of previous ones. This allows to have data inputs to be stored on a low-latency device close to computing nodes and therefore results in reducing the latency of reading data inputs of Big Data applications.

When HPC and Big Data applications are co-located, prefetching operations may interfere with the I/O operations of HPC applications. Thus, our prefetcher component in the burst buffer tries to manage the prefetching of the input data in a controlled manner. Specifically, it takes into consideration the interference at the file system level (i.e., based on the profiled information about the compute and I/O times of both HPC and Big Data applica-

tions which is used for estimating the interference) and makes optimization decisions before each prefetching operation, in order to optimize the performance of Big Data applications while satisfying QoS requirements of HPC applications. Then, it initiates the prefetching operations according to the optimization decisions. We introduce the optimization actions and decision making process of the prefetcher component in the next section.

5.2.2 Design Principles

In this subsection, we present the design principles of Eley which targets improving the performance of Big Data applications on HPC systems without violating the QoS requirements of HPC applications.

Enabling efficient Big Data processing on HPC systems. The separation of storage resources from the compute nodes in HPC systems requires repetitive data transfer through the network, which results in a high I/O latency. In this work, we focus on Big Data applications with huge input data sizes which are executed in multiple waves. Thus, the high I/O latency can severely degrade the performance of Big Data applications on HPC systems. We mitigate the latency problem by locating low-latency burst buffer devices (i.e., RAMDisk) close to the compute nodes to reduce the latency of reading data inputs of Big Data applications.

Existing burst buffer studies mainly focus on mitigating the I/O latency on the intermediate data of Big Data applications. However, as we observed in Chapter 3, using burst buffers for the intermediate data can improve the performance of Big Data applications up to 9%. We also showed that latencies resulting from reading the input data can significantly degrade the performance. Thus, it is also important to mitigate the high I/O latency in the input phase. To this end, we equip our burst buffer with a prefetcher component.

Guaranteeing QoS requirements of HPC applications. Running Big Data applications in HPC systems should not introduce much interference to the HPC applications, which usually involve important scientific simulations. Thus, one of our key design principles is to guarantee the QoS requirements of HPC applications while improving the I/O performance of Big Data applications. We define the QoS requirement of HPC as the deadline constraint for the completion of HPC applications defined by the users. To this end, we equip our prefetcher with interference-aware data transfer scheme to help satisfying our design objective.

5.2.3 Performance Models for HPC and Big Data Applications

The I/O operations of Big Data applications can seriously degrade the I/O performance of concurrent HPC applications. Thus, in this section, we formally model the impact of interference on the I/O time of both Big Data and HPC applications, in order to optimize the performance of Big Data while guaranteeing the QoS constraint of HPC applications.

Performance Model for HPC Applications

Our model focuses on HPC applications with periodical behaviors. That is, the applications perform series of computations and periodically checkpoint the computation results with an I/O time of T_{io}. Most of the HPC applications fall into this category of applications [24]. We assume that the profiles of HPC applications (e.g., I/O and computation time when running individually) can be obtained at offline time using solutions such as Omnisc'IO [38]. We define the I/O time of a HPC application when co-located with a Big Data application as $T_{\text{io}}^{\text{col}}$.

$$T_{\text{io}}^{\text{col}} = T_{\text{io}}^{\text{alone}} + T_{\text{io}}^{\text{intf}} \times I_{\text{hpc}} \tag{5.1}$$

where $T_{\text{io}}^{\text{alone}}$ is the I/O time of the HPC application when performing I/O requests individually and $T_{\text{io}}^{\text{intf}}$ ($T_{\text{io}}^{\text{intf}} = T_{\text{io}} - T_{\text{io}}^{\text{alone}}$) is the I/O time of the HPC application when contending with the Big Data application for I/O resources. I_{hpc} is the interference factor, defined as the ratio between the I/O time of the HPC application under contention and the time for it to perform the same I/O operations alone (i.e., $I_{\text{hpc}} > 1$).

According to Equation 5.1, an important parameter for estimating the I/O performance of HPC applications is the interference factor. The I/O interference imposed by Big Data applications (i.e., prefetching) is affected by several parameters, including the number of the burst buffer nodes performing prefetching (n_{bb}) and the number of fetching threads per node (n_{tr}). I_{hpc} is also depending on the characteristics of the HPC application itself and the platform where it is running. Thus, given an interfering Big Data application to the HPC (i.e., given a set of n_{bb} and n_{tr} values), we can calculate I_{hpc} for the HPC application by profiling its I/O performance with and without contention from the Big Data application. However, offline profiling for each set of given n_{bb} and n_{tr} values is very costly. To reduce the profiling cost, we decompose the interference into two levels, namely node-level (NI) and thread-level (TI) interference, and assume that they are independent from each other. By profiling the two individually, we reduce the profiling complexity from $O(n \times m)$ to $O(n + m)$, supposing that n and m are the sample sizes for n_{bb} and n_{tr}, respectively.

We profile the node-level and thread-level interference and record $NI(n_{bb})$ and $TI(n_{tr})$ values for different n_{bb} and n_{tr} samples. Specifically, we perform prefetching using different numbers of n_{bb}/n_{tr} while fixing n_{tr}/n_{bb} to the default with the HPC application performing checkpointing at the same time. We set the default value of both n_{bb} and n_{tr} to 1. For example, $TI(2)$ is calculated as $\frac{t_2}{t_1}$, where t_1 and t_2 are the I/O performance of the HPC application when one burst buffer node is performing prefetching with 1 and 2 threads, respectively. Based on the above analysis, we can calculate the interference factor for HPC as follows.

$$I_{\text{hpc}}(n_{bb}, n_{tr}) = TI(n_{tr}) \times NI(n_{bb}) \times I_{\text{hpc}}(1,1) \tag{5.2}$$

where $I_{\text{hpc}}(1,1)$ is profiled and calculated at offline time.

Performance Model for Big Data Applications

We provide a performance model for Big Data applications to estimate the prefetching time when using different n_{bb} and n_{tr} values. Similarly to Equation 5.1, the prefetching time is also impacted by the contention duration with the HPC application and the interference factor.

We model the prefetching time when running the Big Data application alone with n_{bb} burst buffer nodes and n_{tr} threads per node as below.

$$T_{\text{pref}}(n_{bb}, n_{tr}) = NS(n_{bb}) \times TS(n_{tr}) \times T_{\text{pref}}(1,1) \tag{5.3}$$

where $NS(n_{bb})$ and $TS(n_{tr})$ represent the system scalability with respect to different number of nodes and number of threads per node, respectively, with the assumption that the node-level and thread-level system scalability are independent. We obtain the scalability values with system profiling using different n_{bb} and n_{tr} samples in the same way as described in the last subsection. For example, $NS(2)$ is calculated as $\frac{t_2}{t_1}$, where t_1 and t_2 are the prefetching time with one and two burst buffer nodes, respectively. Similarly, we obtain $T_{\text{pref}}(1,1)$ by profiling the prefetching time with a single burst buffer node and a single thread.

When co-located with HPC applications, the performance of Big Data applications can be modeled as below.

$$T^{\text{col}}_{\text{pref}} = T^{\text{alone}}_{\text{pref}} + T^{\text{intf}}_{\text{pref}} \times I_{\text{bd}} \tag{5.4}$$

Similarly to Equation 5.1, I_{bd} is the interference factor for the Big Data application, defined as the ratio between the prefetching time with and without co-location with HPC applications. We calculate $I_{\text{bd}}(n_{bb}, n_{tr})$ using offline profiling similarly to Equation 5.2. Specifically, we profile $TI(n_{tr})$ and $NI(n_{bb})$ individually using the prefetching time under different n_{bb} and n_{tr} values.

Furthermore, we define the *cost* of prefetching for Big Data applications as the time that the prefetching cannot be overlapped with the computation phase. To optimize the performance of Big Data applications, we aim to reduce the cost of prefetching in order to hide the I/O latency of reading the input data.

$$C_{\text{pref}} = \begin{cases} T^{\text{col}}_{\text{pref}} - T_{\text{cpu}} & \text{, if } T^{\text{col}}_{\text{pref}} \geq T_{\text{cpu}} \\ 0 & \text{, otherwise} \end{cases} \tag{5.5}$$

where $T^{\text{col}}_{\text{pref}}$ is the time of fetching the input data for the next wave and T_{cpu} is the computation time of the current wave.

5.2.4 Interference-Aware Prefetching

Prefetching of Big Data applications imposes interference to the I/O operations of HPC applications and can violate the QoS requirements of HPC applications. Thus, we propose a set of optimization actions and iteratively choose the best action to optimize the prefetching, in order to satisfy the QoS requirement of the HPC application while leading to a good performance of the Big Data application. In the following, we first introduce the five optimization actions in the action set and then introduce how to choose good actions to optimize prefetching.

Action Set

We propose five optimization actions, including *No Action, Full Delay, Partial Delay, Scale Up* and *Scale Down*. These actions are lightweight and common strategies for interference mitigation. Table 5.1 summarizes the definitions of the five actions. Moreover, we illustrate these

Name	Description
No Action	Start the prefetching as it is.
Full Delay	Delay prefetching to the end of the I/O phase of the HPC application.
Partial Delay	Delay prefetching for a fixed amount of time.
Scale Up	Increase the number of prefetching threads per node.
Scale Down	Decrease the number of total prefetching threads.

Table 5.1: Details of the actions.

actions in Figure 5.3 where the x-axis represents time and y-axis represents the bandwidth of the parallel file system shared between HPC and Big Data applications. The boxes represent the I/O operations of HPC and Big Data applications. The width of boxes represents the completion time and the height represents the acquired bandwidth of the I/O operations. Note that, in the illustration, both applications start I/O operations at the same time. Next, we present the details of the five actions.

No Action performs the prefetching as it is, immediately after receiving the I/O request. We use it as the baseline to show how the other actions optimize the prefetching operation. Thus, the completion time of the I/O operations for Big Data and HPC applications are $T_{\text{pref}} \times I_{\text{bd}}$ and $T_{\text{io}} \times I_{\text{hpc}}$, respectively.

Full Delay delays the prefetching to the end of the I/O phase of the HPC application as shown in Figure 5.3. While this allows HPC application to perform its I/O operations alone and thus mitigates any interference, it can greatly harm the performance of the Big Data application when the HPC application has a long running I/O phase. The completion time of the I/O operations for Big Data and HPC are $T_{\text{pref}} + T_{\text{io}}$ and T_{io}, respectively.

Partial Delay delays the prefetching for a certain amount of time t_d to meet the QoS requirement of the HPC application. We calculate the optimized I/O completion time $T_{\text{io}}^{\text{col}}$ for HPC and $T_{\text{pref}}^{\text{col}}$ for prefetching with t_d delay as follows.

$$T_{\text{io}}^{\text{col}} = t_d + (T_{\text{io}} - t_d) \times I_{\text{hpc}} \tag{5.6}$$

$$T_{\text{pref}}^{\text{col}} = T_{\text{pref}} + (T_{\text{io}} - t_d) \times I_{\text{hpc}} \times (1 - \frac{1}{I_{\text{bd}}}) \tag{5.7}$$

Scale Up increases the number of fetching threads per node. As shown in Figure 5.3, increasing the number of total fetching threads can reduce the prefetching time with an increase on the acquired I/O bandwidth. Therefore, while this action reduces the interference time period of HPC and Big Data applications, it can also lead to a larger performance degradation to HPC due to the higher interference factor I_{hpc}. Assume scale up increases the number of fetching threads per node from n_0 to n_1, we calculate

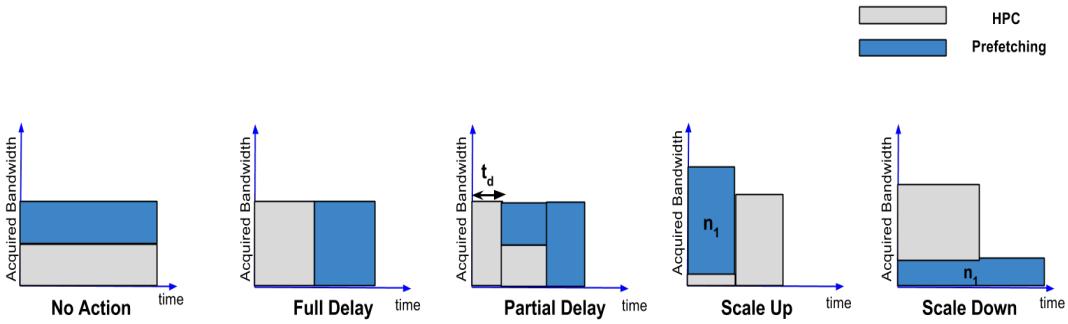


Figure 5.3: Use cases of five actions.

T_{pref}, I_{hpc} and I_{bd} values for n_1 threads based on the model defined in Section 5.2.3 and calculate the optimized I/O completion time for both applications as follows.

$$T_{\text{io}}^{\text{col}} = T_{\text{io}} + T_{\text{pref}} \times I_{\text{bd}} \times (1 - \frac{1}{I_{\text{hpc}}}) \tag{5.8}$$

$$T_{\text{pref}}^{\text{col}} = T_{\text{pref}} \times I_{\text{bd}} \tag{5.9}$$

Scale Down decreases the number of total fetching threads from n_0 to n_1. In the previous chapter, we observed that constraining the rate that Big Data applications send their I/O requests can help to lower the interference. Hence, using less fetching threads can diminish the impact of prefetching on the HPC application with a smaller interference factor (I_{hpc}) but it can also lead to a significant poor I/O performance of the Big Data application due to the increased prefetching time (T_{pref}). When n_1 is smaller than the number of burst buffer nodes, we use n_1 nodes and one thread per node for the prefetching. Similarly to scale up, we calculate T_{pref}, I_{hpc} and I_{bd} values for n_1 threads, and calculate the optimized I/O completion time for both applications as follows.

$$T_{\text{io}}^{\text{col}} = T_{\text{io}} \times I_{\text{hpc}} \tag{5.10}$$

$$T_{\text{pref}}^{\text{col}} = T_{\text{pref}} + T_{\text{io}} \times I_{\text{hpc}} \times (1 - \frac{1}{I_{\text{bd}}}) \tag{5.11}$$

Decision Making

Given the optimization actions, we iteratively decide the optimal action for prefetching at the beginning of each iteration (e.g., wave) of the Big Data application, in order to reduce the cost of the Big Data application while satisfying the performance constraint of the HPC application.

When making prefetching decisions, we estimate the cost of the Big Data application and the I/O time of the HPC application for each action. For example, when estimating the cost of the partial delay action, we first calculate the completion time of both HPC and Big Data applications with different t_d values using Equation 5.6 and 5.7. We then select the t_d that leads to the minimum cost of the Big Data application while satisfying the deadline

Iteration	Condition	Action
1	No action necessary for satisfying the HPC constraint	No Action
2	Full Delay has zero cost for the Big Data application while satisfying the HPC constraint	Perform Full Delay
3	Partial Delay has zero cost for the Big Data application while satisfying the HPC constraint	Perform Partial Delay
4	Scale Up has zero cost for the Big Data application while satisfying the HPC constraint	Perform Scale Up
5	Scale Down has zero cost for the Big Data application while satisfying the HPC constraint	Perform Scale Down
6	None of the actions has zero cost	Perform an action which has the minimum cost

Table 5.2: Iterations in decision making when choosing the optimal action.

constraint of the HPC application for the delay action and return its result as the result of partial delay. If we can find an action with a cost equal to zero for the Big Data application while satisfying the deadline constraint for the HPC application, we simply adopt this action without further checking the other actions. If none of the actions achieves zero cost for the Big Data application, we choose the action with the minimum cost while satisfying the deadline constraint of the HPC application. Table 5.2 summarizes this iterative decision making process for choosing the optimal action.

5.3 Experimental Evaluation

5.3.1 Methodology

We evaluate our burst buffer design using both real system experiments and simulations. We further describe the experimental setup for both the real system evaluation and the simulations.

Real System Setup

The empirical experiments were carried out on the Grid'5000 testbed [19]. We used the Rennes site; more specifically we employed nodes belonging to the *parasilo* and *paravance* clusters. The nodes in these clusters are outfitted with two 8-core Intel Xeon 2.4 GHz CPUs and 128 GB of RAM. We leveraged the 10 Gbps Ethernet network that connects all nodes of these two clusters. Grid'5000 allows us to create an isolated environment in order to have full control over the experiments and obtained results.

System deployment. We used Spark version 1.6.1 to execute Big Data applications. We configured and deployed a Spark cluster using 33 nodes on the *paravance* cluster. One node consists of the Spark master, leaving 32 nodes to serve as slaves of Spark. We allocated 8 GB per node for the Spark instance and set Spark's default parallelism parameter (spark.default.parallelism) to 256 which refers to the number of RDD partitions. Each Spark slave has 16 map tasks thus the Spark cluster can execute 512 map tasks in one iteration.

In our burst buffer design, we use low-latency storage devices as a storage space. To emulate this, we used Alluxio version 1.3.1, which exposes RAMDisk of the burst buffer nodes to the compute nodes as an in-memory file system. We configured and deployed 8 burst buffer nodes on the same cluster (*paravance*) as compute nodes to emulate that burst buffer nodes are closer to compute nodes compared to the parallel file system. Each burst buffer node provides approximately 32 GB of storage capacity.

The OrangeFS file system (a branch of PVFS2 [110]) version 2.8.3 was deployed on 6 nodes of the *parasilo* cluster to serve I/O requests from both Big Data and HPC applications. We select 6 PVFS nodes using the same setting as existing studies [144].

Workloads. We selected two workloads including Sort and Wordcount, which are parts of the HiBench [58], a Big Data benchmarking suite. Wordcount is a map-heavy workload with a light reduce phase. On the other hand, Sort produces a large amount of intermediate data which leads to a heavy shuffling, therefore representing reduce-heavy workloads. For both workloads, we use 160 GB of input data generated by RandomTextWriter benchmark of the HiBench suite. Both workloads are executed with five iterations, where each iteration processes input data size of 32 GB.

As HPC workloads, we use IOR [119] which is a popular I/O benchmarking tool for HPC systems. We choose this workload for controllable interference between Big Data and HPC to evaluate the effectiveness of Eley. For each set of experiment, we run IOR side by side with the Big Data workloads using the same number of iterations. In each iteration, we use IOR to emulate the I/O requests of HPC applications. Between each request, IOR processes sleep for a given time S to emulate the computation time of HPC applications. We set S to be equal to the computation time (T_{cpu}) of Big Data applications.

Simulation Setup

We design a deterministic event-driven simulator to simulate the execution of HPC and Big Data applications in a system with the same configuration as our real deployment. We choose three write-intensive scientific applications executed on Intrepid in 2011 [88] as the HPC workloads. Table 5.3 shows the characteristics of those applications. For Big Data applications, we use the traces collected from a OpenCloud cluster in Carnegie Mellon University [52]. Table 5.4 presents the characteristics of the eight applications used in our simulation, which are scientific applications that are most likely to be co-located with HPC applications in HPC systems.

We run HPC applications for 10 iterations in the simulation. For Big Data applications, the number of iterations (waves) depends on the total input size and the cluster capacity. We define the cluster capacity as 50 GB in our simulation. Thus we can execute 800 map tasks in one iteration. From our real system evaluation, we observe that the bandwidth of the par-

Application	Compute Time(s)	I/O Volume(GB)
Turbulence1 (T1)	70	128.2
Turbulence2 (T2)	1.2	235.8
AstroPhysics (AP)	240	423.4

Table 5.3: HPC application characteristics for the simulation [88].

Application	Compute Time(s)	Input Data Size(GB)
1	20	195
2	24	860
3	12	600
4	236	355
5	29	662
6	43	799
7	41.5	1300
8	9.5	13000

Table 5.4: Big Data application characteristics for the simulation.

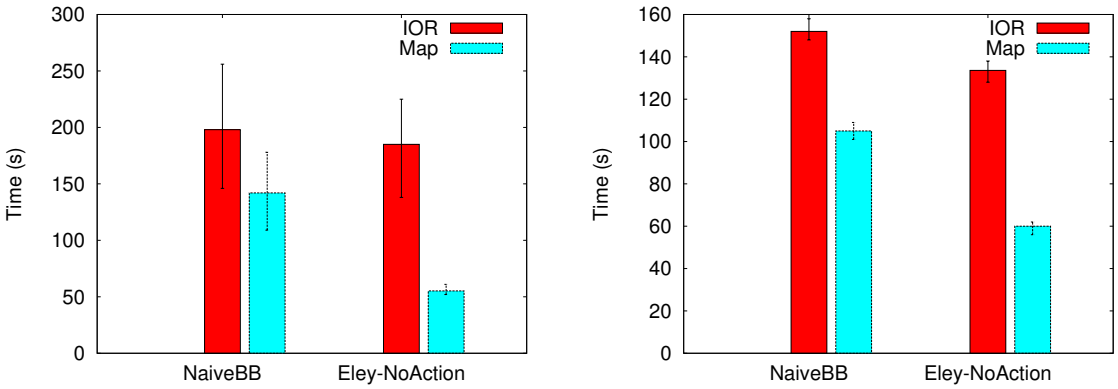
allel file system is 4 GB/s. Thus, we use this value in our simulation to limit the maximum I/O bandwidth acquired by Big Data and HPC applications to the parallel file system. Based on the model defined in Section 5.2.3, we generate I/O requests (i.e., interference factor, data copying/fetching times) by using the profiling values obtained during our real system experiments. This profiling approach, which needs to be performed only once for a particular hardware, helps us to emulate the I/O behaviors of Big Data and HPC applications which differ with the compared burst buffer solutions. We perform a set of microbenchmarks to favor the validity of the simulation results on Grid'5000 testbed. However, the validity of the simulation results on other platforms than Grid'5000 remains to be investigated.

Comparisons

For both real system and simulator-based experiments, we adopt the following state-of-the-art burst buffer solutions as comparisons to *Eley*.

NaiveBB. As in the existing burst buffer solutions [25, 73, 114, 139], *NaiveBB* uses burst buffer only for storing the intermediate data of Big Data applications. While it is already shown that this approach can improve the performance of Big Data applications, different from *Eley*, it does not consider the QoS requirements of HPC applications and the I/O latency problem in the input phase.

Eley-NoAction. This approach performs naive prefetching to copy the input data of Big Data applications from the parallel file system to burst buffer nodes. Different from *Eley*, this approach is not aware of the QoS requirement of HPC applications and does not use any optimization action for the prefetching.



(a) Sort. (b) Wordcount.

Figure 5.4: Performance comparison between Eley-NoAction and NaiveBB.

All solutions are evaluated using the same burst buffer storage space (i.e., RAMDisk) for a fair comparison. We evaluate the completion time of both Big Data and HPC applications obtained by the three compared solutions to demonstrate the effectiveness of Eley. For Big Data applications, we only report the completion time of the map phase which is the main differentiation between Eley and the compared solutions. We also compare the resulted interference factors of both applications after applying the three compared burst buffer solutions, which indicate the slowdown (defined as $I - 1$) of the applications due to co-location and I/O contention.

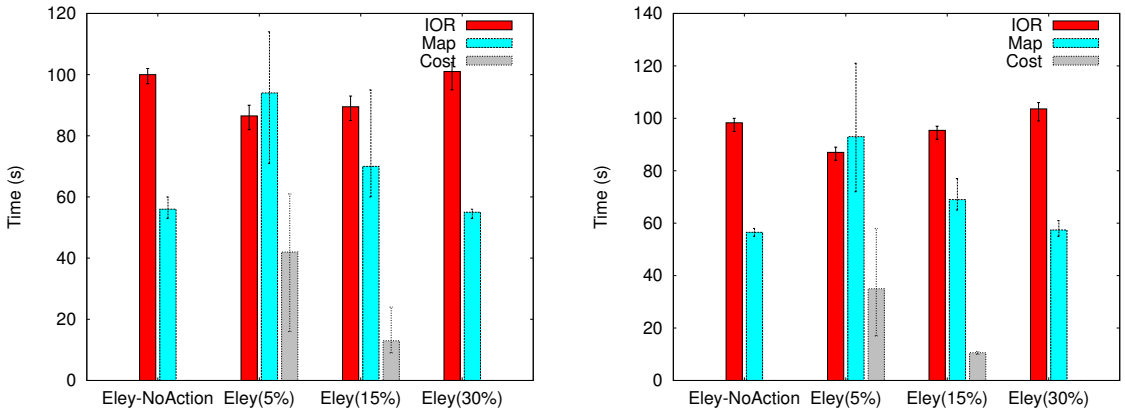
5.3.2 Real System Experiments

In this section, we present the evaluation results obtained from both real system deployment and simulations.

We perform two sets of real system experiments. First, we compare *Eley-NoAction* with *NaiveBB*, the results demonstrate that using a burst buffer for prefetching the input data of Big Data applications can greatly improve the performance of Big Data while reducing the interference imposed on the co-located HPC applications. Second, we compare *Eley* with *Eley-NoAction* and find that our interference-aware model and optimization actions can successfully guarantee the QoS requirement of HPC applications without sacrificing too much of the Big Data performance. Each set of experiments has been executed five times and we report the average values.

Comparing *NaiveBB* with *Eley-NoAction*

In this experiment, we execute IOR on a cluster of 32 nodes where one process per node issues a 4 GB write request with chunk size of 1MB. Figure 5.4 shows that prefetching reduces the map time of Sort and Wordcount by 61% and 43%, respectively, compared to *NaiveBB* which reads the input data from the parallel file system directly. It can be observed that the I/O time of the IOR workloads are also reduced when using prefetching for Big Data, by 7% and 12% when co-locating with Sort and Wordcount, respectively. This is mainly due to the



(a) Sort. (b) Wordcount.

Figure 5.5: Performance comparison between Eley and Eley-NoAction.

	Iter 1	**Iter 2**	**Iter 3**	**Iter 4**	**Iter 5**
IOR Time (s) (*Eley-NoAction*)	17.8	22.4	18.5	20.9	23
IOR Time (s) (*Eley*)	17.8	17.6	17.1	16.3	17.3
Selected Action	No Action	Partial Delay	Scale Up	Partial Delay	Full Delay
Cost of Sort (s)	0	13	0	9	15

Table 5.5: Applied actions on prefetching during one run of the Sort application and their respective cost values.

fact that prefetching using burst buffer aggregates the I/O requests sent from a large number of Spark nodes to a small set of burst buffer nodes and hence reduces the interference imposed on IOR workloads. For instance, while 32 compute nodes request the I/O from PVFS in *NaiveBB*, only 8 burst buffer nodes perform prefetching with *Eley-NoAction*. This is inline with our observations in Chapter 4 and the existing studies in the literature [1, 36] which demonstrate that aggregation of I/O requests help to reduce the I/O interference. However, this small improvement might not be enough to meet the QoS requirements of HPC applications. Therefore, we next discuss the effectiveness of our interference-aware prefetching mechanism, *Eley*.

Comparing *Eley-NoAction* with *Eley*

In this experiment, we execute IOR on a cluster of 8 nodes, where one process per node issues a 8 GB write request with chunk size of 1MB. We set three different deadline requirements for IOR workloads, which are 5%, 15% and 30% longer than the completion time of the workloads when executed individually.

Figure 5.5 shows that *Eley* is aware of the deadline constraints of HPC applications and can control the level of interference imposed by Big Data to HPC applications accordingly. We take a closer look at the optimization actions used in each iteration to evaluate the effectiveness of *Eley* on guaranteeing the QoS of HPC applications. Table 5.5 shows the applied

Application	NaiveBB (s)	Eley-NoAction (s)
1	151.7	136.6
2	771.3	526
3	379	358.6
4	2018	1925
5	674	487.3
6	979.7	770
7	1518.7	1174
8	6239	6686.7

Table 5.6: Average map times of Big Data applications with NaiveBB and Eley-NoAction approaches when running together with the three HPC applications.

actions during one run of the Sort application with a QoS of 5% (i.e., 17.9 s). We can observe that with the optimization actions, *Eley* is always able to guarantee the QoS requirement with low cost of the Big Data application. For example, in iteration 3, *Eley* is able to find an action (i.e., Scale Up) which leads to a zero cost for the Sort application.

The selection of actions depends on both the characteristics of Big Data and HPC applications. For instance, the Scale Down action is not used during the entire execution of the Sort application. We believe this is mainly due to the relatively short computation time of Sort compared to its prefetching time (e.g., 12 seconds and 16 seconds per iteration, respectively).

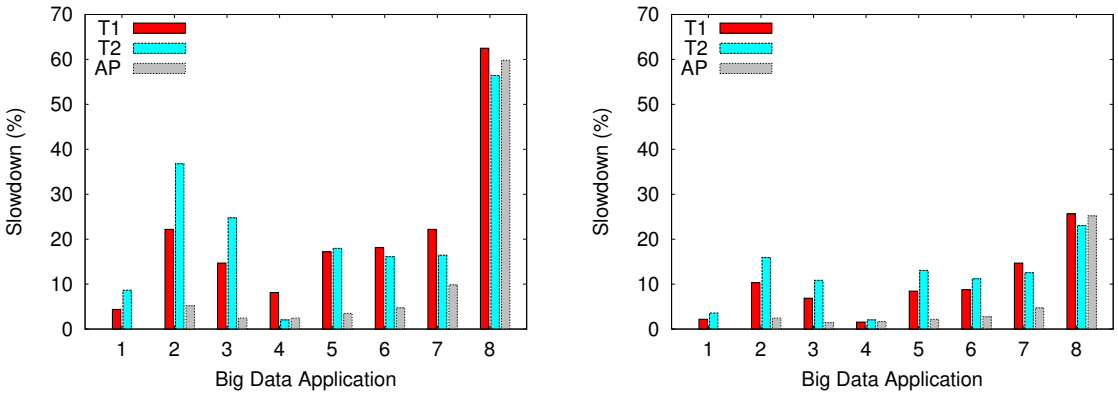
Another observation from Figure 5.5 is that there is a clear trade-off between the tightness of HPC deadline and the performance of Big Data applications. For example, with the 5% QoS requirement, the cost of Sort and Wordcount applications are 60% and 75%, respectively. The cost reduces to 23% and 21% for Sort and Wordcount, respectively, when the QoS requirement is relaxed to 15%. With a small cost, the performance of the Big Data applications are improved. The cost is also depending on several factors, such as the characteristics of HPC and Big Data applications and the HPC platform on which the applications are executing. In the next subsection, we run a wide range of applications with our simulator to better study this trade-off.

5.3.3 Simulation Results

Similarly to the real system experiments, we perform two sets of comparisons using the three burst buffer solutions.

Comparing *NaiveBB* with *Eley-NoAction*

Table 5.6 shows the average map time of the eight Big Data applications when running together with the three HPC applications. We can observe that *Eley-NoAction* can reduce the map time of Big Data applications (except Application 8) by up to 32% compared to *NaiveBB*. *Eley-NoAction* achieves performance improvement for Big Data applications mainly by overlapping the input data reading time and map computation time. However, as Application 8 has a very small computation time compared to the required prefetching time, it cannot benefit much from the *Eley-NoAction* solution. The prefetching time for the application is



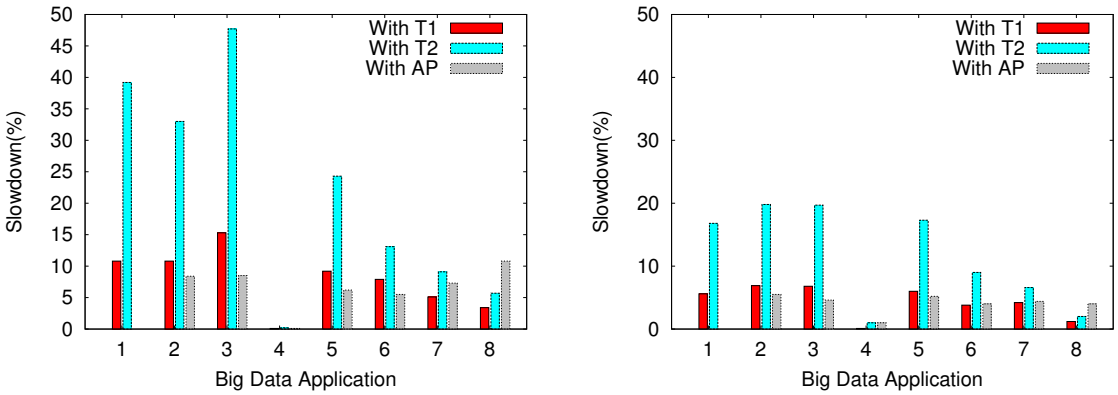
(a) NaiveBB. (b) Eley-NoAction.

Figure 5.6: Slowdowns observed for the three HPC applications with NaiveBB and Eley-NoAction.

25 seconds while reading the input data directly from the parallel file system is only 12 seconds. This is mainly due to the different number of nodes sending I/O requests in *naiveBB* and *Eley-NoAction*. On the other hand, we find that the improvement on the performance of Big Data applications brought by prefetching is small when the application has a very large computation time. For example, the map computation time of Application 4 is 236 seconds per iteration, which is 10 times larger than its prefetching time. Thus, the performance improvement for this application when using *Eley-NoAction* is only 5%.

We evaluate the slowdown ($I_{\text{hpc}} - 1$) of the three HPC applications when running together with the eight Big Data applications using *NaiveBB* and *Eley-NoAction* approaches as shown in Figure 5.6. We have the following observations. First, when using *NaiveBB*, most of the HPC applications suffer a lot from the I/O interference. For example, T1 has the highest slowdown of 63% when co-locating with Big Data Application 8. Second, the slowdown of the HPC applications are greatly reduced by *Eley-NoAction* compared to *NaiveBB*. For example, the slowdown of T1 is reduced to 26% when running with Application 8 using *Eley-NoAction*. Third, the reduction of slowdown varies from application to application. For example, T1 achieves the highest reduction of slowdown with 54% on average while the reduction of slowdown for T2 and AP are both 39% on average. We also observe that Application 8 is imposing the highest slowdown to all HPC applications due to its short computation time and frequent I/O requests.

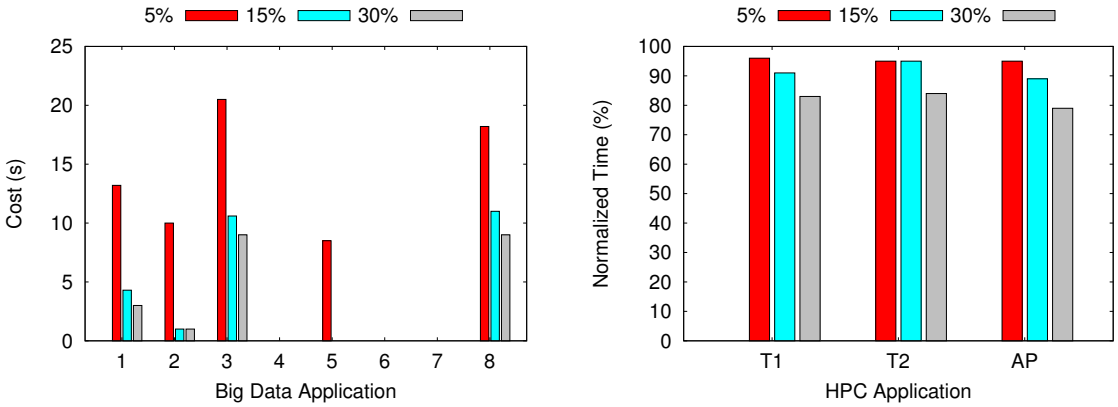
We further study the slowdown ($I_{\text{bd}} - 1$) of the Big Data applications using the two burst buffer solutions as shown in Figure 5.7. We have similar observations as those for HPC applications. First, the *NaiveBB* solution introduces a high slowdown to Big Data applications, e.g., 48% for Application 3 when co-located with T2. Second, *Eley-NoAction* can greatly reduce the slowdown of Big Data applications compared to *NaiveBB*. For example, the reduction of slowdown can reach up to 58% on average for Application 3. Third, T2 is the HPC application which imposes the highest slowdown to the Big Data applications. This is again due to the fact that T2 has a small computation time and thus its I/O operations can interfere frequently with the I/O operations of Big Data applications.



(a) NaiveBB.

(b) Eley-NoAction.

Figure 5.7: Slowdowns observed for the Big Data applications with NaiveBB and Eley-NoAction.



(a) Average cost for the Big Data applications.

(b) Normalized I/O time of HPC applications respect to QoS.

Figure 5.8: Performance of Big Data and HPC applications with Eley approach.

Comparing *Eley-NoAction* with *Eley*

We study the effectiveness of *Eley* on satisfying the QoS requirements of different HPC applications with different concurrently running Big Data applications. We set three QoS requirements for HPC applications, which are 5%, 15% and 30% longer than the completion time of the applications when executed individually. Figure 5.8(a) shows the average cost of the Big Data applications and Figure 5.8(b) shows the average performance of the HPC applications normalized to the different QoS requirements when optimized with *Eley*.

Figure 5.8(a) shows that the cost of Big Data applications decreases with more relaxed QoS requirements. For Big Data applications with longer prefetching time than computation time, the initial cost before applying any action is high. For example, Applications 3 and 8 have the highest cost values due to their relatively short computation times. Figure 5.8(b) shows that *Eley* is able to satisfy the QoS requirement for the three HPC applications under

	Iter 1	Iter 2	Iter 3	Iter 4	Iter 5
T2 I/O Time (s) (*Eley-NoAction*)	70	65	68	65	69
T2 I/O Time (s) (*Eley*)	59	59	62	59	59
Selected Action	Scale Down	Full Delay	Partial Delay	Scale Down	Scale Down
Cost of Application 6 (s)	6	0	0	6	6

Table 5.7: Applied actions on prefetching during the first 5 iterations of the Application 6 and T2 and their respective cost values.

all settings. This demonstrates that our interference-aware mechanism is effective in guaranteeing the QoS for HPC applications. Take the execution of Big Data Application 6 with the T2 application for example. Table 5.7 shows the optimization actions selected when the QoS requirement of T2 is set to 5% (i.e., 62 seconds). For all the iterations, *Eley* is able to satisfy the QoS requirement of T2. In two of the iterations, we are able to find actions leading to zero cost of the Big Data application. Different from the real system experiment, we find that the Scale Down action is selected at the last two iterations. This is mainly because Application 6 has almost twice longer computation time compared to its prefetching time (recall that Sort has shorter computation time than its prefetching time and thus never uses Scale Down).

5.3.4 Summary of the Results

In Chapter 3, we highlighted that alleviating I/O latency and interference is crucial toward performing efficient Big Data processing on HPC systems. Our results demonstrate that Eley effectively achieves this and improves the performance of Big Data applications while meeting the QoS requirements of HPC applications (expressed as deadline constraints). This is due to two main factors:

- *Overlapping I/O and computation:* Thanks to its *prefetcher* component, Eley improves the performance of Big Data applications by overlapping I/O and computation time. Eley achieves this by copying the input data from the parallel file system to the burst buffer nodes while computing nodes are still busy processing the input data of previous ones. Thus, compute nodes can read the input data from burst buffer nodes which offer high throughput I/O operations, on the contrary to the parallel file system. We also showed that this performance improvement brought by prefetching depends on the characteristics of Big Data applications. For instance, when the application has a very small computation time Eley would not be able to fully hide the cost of I/O latency when reading the input data.

- *OoS-aware execution:* HPC applications usually involve important scientific objectives thus it is important to meet the QoS requirements of HPC applications despite co-located Big Data applications. We showed that Eley can guarantee the QoS requirements of HPC applications by employing an interference-aware prefetching mechanism. We also observed that existing burst buffer solutions (denoted as NaiveBB) can significantly violate such QoS requirements.

5.4 Related Work

5.4.1 Early Adoption of Big Data Processing in HPC Systems

Many research efforts have been dedicated to adopt Hadoop [53] on HPC systems [93, 126, 143]. Researchers have discussed solutions to extend data locality to parallel file systems (i.e., PVFS [110]) through emulating HDFS [55] on a HPC system by using PVFS [126] or proposing a new storage layer (in-memory storage) on the top of PFVS [143]. Those solutions assume that computing nodes have dedicated disk, this however is not a typical configuration of HPC system. Moreover, new MapReduce frameworks — beyond Hadoop — have been introduced such as MARIANE [40] and Glasswing [57]. They mainly focus on exploiting multi-core CPU and GPUs. Although aforementioned works can improve the performance of Big Data applications compared to a blind adoption of these applications on HPC systems, they do not fully address the challenges (i.e., interference, high latency) and thus they are not able to efficiently leverage HPC systems for Big Data processing.

5.4.2 Extending Burst Buffers for Big Data Applications in HPC Systems

Burst Buffers are an effective solution for reducing the I/O latency and the I/O interference in HPC systems. Kougkas et al. [79] leveraged burst buffers when coordinating the I/O requests of contending HPC applications. In particular, they dynamically partitioned the parallel file system resources among applications and delayed the I/O requests without interrupting the computation by using burst buffers. Thapaliya et al. [127] proposed I/O scheduling policies at the burst buffer level to mitigate the cross-application I/O interference. Wang et al. [138], leveraged burst buffers to reduce the I/O latency in storing the checkpointing data of HPC applications and thus improving the I/O performance.

Hence, several works proposed adoption of burst buffers for an efficient Big Data processing on HPC systems. Chaimov et al. [25] employed a NVRAM buffer between compute nodes and Lustre file system in order to improve the scalability of the Spark framework. Islam et al. [72] proposed using Memcached as a burst buffer system to integrate HDFS with Lustre file system in a performance-efficient way. They also evaluated the NVRAM performance as a burst buffer system in [73]. Wang et al. [139] performed an experimental study where they investigated the characteristics of Spark on a HPC system with a special focus on the impact of the storage architecture. Based on their findings, they proposed to use SSDs as a burst buffer to store the intermediate data. As we demonstrated, however, although these studies are indeed important — without considering the interference problem and the latency resulting from the input phase — they do not provide a complete solution for enabling efficient Big Data processing on HPC systems.

5.5 Conclusions

In this work, we propose *Eley*, a burst buffer solution which takes into consideration both the input data and intermediate data of Big Data applications. Our goal is to accelerate the performance of Big Data applications while guaranteeing the QoS of HPC applications. To achieve this goal, *Eley* is composed of a prefetcher, which prefetches the input data of Big

Data applications before the execution of each iteration. By prefetching, we are able to overlap the I/O and computation time to improve the performance of Big Data applications. However, data prefetching may introduce huge I/O interference to the HPC applications and thus end up with a degraded and unpredictable performance for HPC applications. To this end, we design *Eley* to be interference-aware. Specifically, we equip the prefetcher with five optimization actions and propose an interference-aware decision maker to iteratively choose the best action to optimize the prefetching while guaranteeing the pre-defined QoS requirement of HPC applications. We evaluate the effectiveness of *Eley* with both real system experiments and simulations. With 5% QoS requirement of the HPC application, *Eley* reduces the execution time of Big Data applications by up to 23% compared to the naive burst buffer solution (denoted as *NaiveBB*) [25, 73, 114, 139] while guaranteeing the QoS requirement. On the other hand, the *NaiveBB* violates the QoS requirement by up to 58%.

Chapter 6

Chronos: Enabling Fast Failure Recovery in Large Shared Clusters

Contents

6.1 The Unawareness of Failures in Current Job Schedulers 80
6.2 Fast Failure Recovery with Failure-Aware Scheduling 81
6.2.1 Chronos: Design Principles 81
6.2.2 Work-conserving Preemption 83
6.3 Experimental Evaluation 87
6.3.1 Methodology 88
6.3.2 The Effectiveness of Chronos in Reducing Job Completion Times . . 89
6.3.3 The Effectiveness of Chronos in Improving Data Locality 90
6.3.4 Impact of Reduce-Heavy Workloads 91
6.3.5 The Effectiveness of the Preemption Technique 91
6.3.6 Overhead of Chronos 92
6.3.7 Chronos and Hadoop Under Multiple Failures 92
6.3.8 Chronos with Aggressive Slot Allocation 92
6.3.9 Discussion of the Results 94
6.4 Related Work 95
6.4.1 Scheduling in Big Data Systems 95
6.4.2 Exploiting Task Preemption in Big Data Systems 95
6.5 Conclusions 96

FAILURES are part of everyday life, especially in today's large-scale platforms, which comprise thousands of hardware and software devices. For instance, Dean [29] reported that in the first year of the usage of a cluster at Google there were around a

thousand individual machine failures and thousands of hard drive failures. Schroeder et al. analyzed the failure rates in the machines at Los Alamos National Laboratory and reported that there were systems with more than 1100 failures per year [116]. Since failures are more severe in clouds due to employing failure-prone commodity machines, we focus on the failure handling in clouds in this chapter.

Failures result in a severe performance degradation for Big Data applications. This is because the completion time of an application, which indicates its performance, is determined by its slowest task. Consequently, the completion time of the recovery tasks will be the major factor for the performance of an application. For instance, Dinu et al. reported that the performance of one single Big Data application degrades by up to 3.6X for one single machine failure [32]. This problem is more severe when multiple applications running concurrently on clouds due to a higher competition in obtaining the resources for the recovery tasks. To mitigate the impact of failures on the performance of Big Data applications, we propose the Chronos[1] scheduler in this chapter. Different from the state-of-the-art schedulers where failure handling is entrusted to the core of Big Data processing frameworks, Chronos is a failure-aware scheduling strategy that enables an early yet smart action for fast failure recovery while operating within a specific scheduler objective. Moreover, Chronos considers I/O latency by aiming at sustaining a high data locality even under failures. We implemented Chronos in Hadoop and demonstrated its utility by comparing it with two state-of-the-art Hadoop schedulers: Fifo and Fair schedulers.

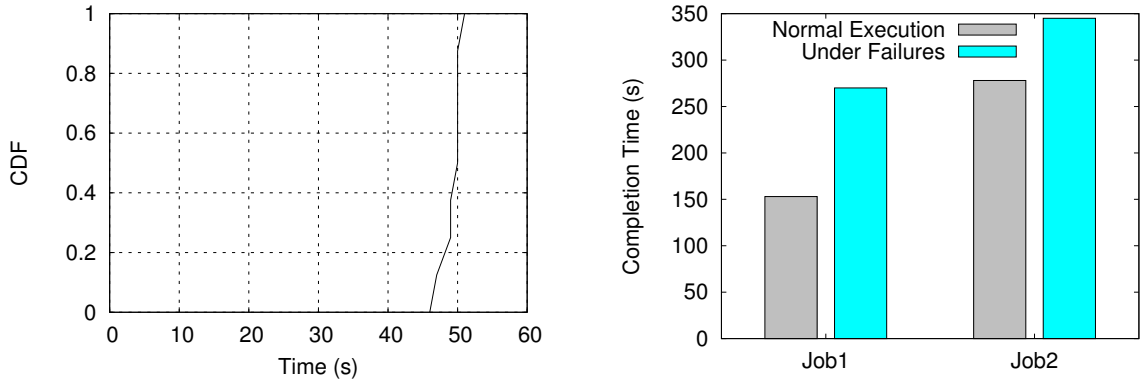
6.1 The Unawareness of Failures in Current Job Schedulers

Several built-in schedulers (i.e., Fifo, Fair and Capacity schedulers) have been introduced in Big Data processing frameworks to operate shared Big Data clusters towards a certain objective (i.e., prioritizing jobs according to their submission times in Fifo scheduler; favoring fairness among jobs in Fair and Capacity schedulers) while ensuring a high performance of the system. This is done through accommodating these schedulers with locality-oriented strategies [6, 59, 150]. In particular, these schedulers adopt a resource management model in Hadoop which is based on *slots* (i.e., slots typically correspond to the cores in Hadoop workers) to represent the capacity of a cluster: each worker in a cluster is configured to use a fixed number of map slots and reduce slots in which it can run tasks.

When failures are detected, in order to launch recovery tasks, empty slots are necessary. If the cluster is running with the full capacity, then Hadoop has to wait until “free" slots appear. However, this waiting time (i.e., time from when failure is detected until all the recovery tasks start) can be long, depending on the duration of current running tasks. As a result of this *uncertain waiting time*, the following problems may arise:

- A violation of scheduling objectives is likely to occur (e.g., high-priority jobs may have waiting tasks while lower priority jobs are running) and the performance may significantly degrade.
- When launching recovery tasks, data locality is totally ignored. This in turn can further increase the job completion time due to the extra cost of transferring a task’s input data through network, a well-known source of overhead in today’s large-scale platforms.

[1] From Greek philosophy, the god of time.



(a) CDF of waiting times of recovery tasks

(b) Job completion times

Figure 6.1: Recovery task waiting times and job completion times of Fifo scheduler under failure. The experimental setup is the same as in Section 6.3.2.

These observations present an opportunity for taking early actions upon failure detection rather than waiting an uncertain amount of time and incurring performance degradation. On this base, we introduce our Chronos scheduler which enables fast failure recovery in shared Hadoop clusters.

6.2 Fast Failure Recovery with Failure-Aware Scheduling

Adding failure-awareness to Hadoop schedulers is not straightforward; it requires the developer to carefully deal with challenging yet appealing issues including an appropriate selection of slots to be freed, an effective preemption mechanism with low overhead and enforcing data locality for the execution of recovery tasks. To the best of our knowledge, no scheduler explicitly coping with failures has been proposed. In this section, we discuss how Chronos achieves these goals.

6.2.1 Chronos: Design Principles

To enable an efficient fast failure recovery, we designed Chronos with the following goals in mind:

Enabling an early action upon failure. Hadoop handles failures by scheduling recovery tasks to any available slots. However, available slots might not be freed up as quickly as expected. Thus, recovery tasks will be waiting an uncertain amount of time which depends on the status of running tasks (i.e., current progress and processing speed) when failure is detected. As shown in Figure 6.1(a), the waiting time varies from *46* to *51 seconds* which leads to increase in the completion time of jobs (see Figure 6.1(b)). Furthermore, during this time, scheduling objectives are violated. Chronos thus takes immediate action to make room for recovery tasks upon failure detection rather than waiting an uncertain amount of time.

Minimal overhead. For the early action, a natural solution is to kill the running tasks in order to free slots for recovery tasks. Although the killing technique can free the slots easily, it results in a huge waste of resources: it discards all of the work performed by the killed tasks. Therefore, Chronos leverages a work-conserving task preemption technique (Section 6.2.2) that allows it to stop and resume tasks with almost zero overhead.

Data-local task execution. Although data locality is a major focus during failure-free periods, locality is totally ignored by Hadoop schedulers when launching recovery tasks (e.g., our experimental result reveals that Hadoop achieves only 12.5% data locality for recovery tasks, more details are given in Section 6.3.3). Chronos thus strongly considers local execution of recovery tasks.

Performance improvement. As shown in Figure 6.1(b), the job completion times increase by 30% to 70%. Through eliminating the waiting time to launch recovery tasks and efficiently exposing data-locality, Chronos not only corrects the scheduling behavior in Hadoop after failure but also improves the performance.

Resource utilization. Chronos aims at having better resource utilization by two key design choices. First, it reduces the need for remote transfer of the input data by launching local recovery tasks. Second, Chronos uses a work-conserving preemption technique that prevents Chronos from wasting the work done by preempted tasks.

Schedulers independent. Chronos targets to make Hadoop schedulers failure-aware and is not limited to Fifo or Fair schedulers. Taken as a general failure-aware scheduling strategy, Chronos can be easily integrated with other scheduling policies (e.g., priority scheduling with respect to the duration of jobs). Moreover, our preemption technique can be used as an alternative solution to task killing or waiting and therefore can leverage the scheduling decision in Hadoop in general.

Hereafter, we will explain how Chronos achieves the above goals. We will discuss how Chronos allocates the necessary slots to launch recovery tasks, thanks to the tasks-to-preempt selection algorithm.

Smart slots allocation

Figure 6.2 illustrates the architecture of Chronos and its main functionalities. Chronos tracks the progress of all running tasks using the cost-free real-time progress reports extracted from the heartbeats. Here, Chronos waits until new hearbeats are received in order to have up-to-date information. Relying on previous heartbeats information may result in preempting almost-complete tasks. This waiting time, introduced by Chronos, ranges from milliseconds to several seconds, according to the heartbeat interval and the current network latency. Hadoop adjusts the heartbeat interval according to the cluster size. The heartbeat interval is calculated so that the JobTracker receives a *min_heartbeat_interval* of 100 heartbeat messages every second. The heart beat interval in Hadoop is computed as:

$$heartbeat_interval = \max((1000 \times cluster_size \div number_heartbeat_per_second), min_heartbeat_interval) \quad (6.1)$$

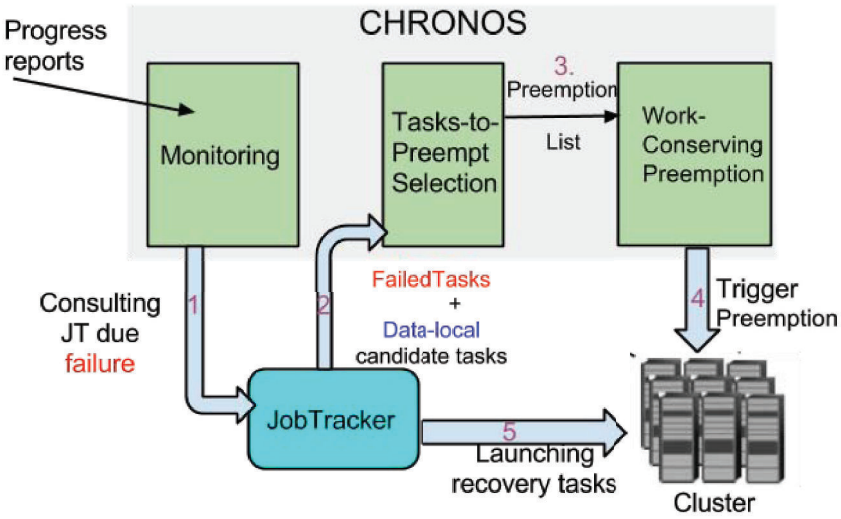


Figure 6.2: Chronos overview.

Importantly, the *min_heartbeat_interval* limits the heartbeat interval to not be smaller than 300 milliseconds. In our experiments, the heartbeat interval was *0.3 seconds* corresponding to the cluster size.

When some failure is detected, Chronos consults the JobTracker to retrieve the list of failed tasks and the nodes that host their input data. Chronos then extracts the list of candidate tasks (running tasks) that belong to nodes where the input data of failed tasks reside. This list is then fed to the tasks-to-preempt selection algorithm (*Algorithm 1*) which first sorts the tasks according to the job priority. After sorting the tasks for preemption, the next step is to decide whether a recovery task can preempt any of these tasks in the sorted list. To respect scheduling objectives, we first compare the priorities of the recovery task and candidate tasks for preemption. If this condition holds, the recovery task can preempt the candidate task. For example, recovery tasks with higher priority (e.g., tasks belonging to earlier submitted jobs for Fifo or belonging to a job with a lower number of running tasks than its fair share for Fair scheduler) would preempt the selected tasks with less priority. Consequently, Chronos enforces priority levels even under failures. The list is then returned to Chronos, which in turn triggers the preemption technique in order to free slots to launch recovery tasks. If the scheduler behavior is corrected, Chronos stops preempting new tasks.

6.2.2 Work-conserving Preemption

Preemption has been widely studied and applied for many different use cases in the area of computing. Similarly, Hadoop can also benefit from the preemption technique in several cases (e.g., achieving fairness, better resource utilization or better energy efficiency). However, only the kill technique (i.e., killing the tasks) is available in Hadoop which can be used by developers as a preemption technique. Although the kill technique is simple and can take a fast action (i.e. deallocating the resources), it wastes the resources by destroying the ongoing work of the killed tasks. This amount of wasted work will even increase with the long

Algorithm 1: Tasks-to-preempt Selection Algorithm

Data: L_{tasks}, a list of running tasks of increasing priority;
T_r, a list of recovery tasks t_r of decreasing priority
Result: T_p, a list of selected tasks to preempt

```
1 for Task t_r:T_r do
2     for Task t:L_tasks do
3         if t belongs to job with less priority compared to t_r;
4          AND !T_p.contains(t) then
5             T_p.add(t);
6         end
7     end
8 end
```

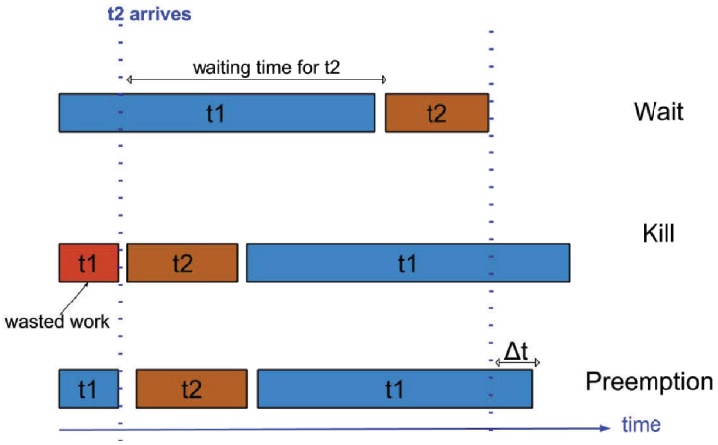


Figure 6.3: Overview of the three preemption techniques.

running tasks. Moreover, this amount can be more significant if the killed task is a reduce task: later, when the copy of the killed task is launched, it has to fetch all the intermediate mapper outputs which will result in additional usage of network resources that have already been scarce.

Apart from killing, also a waiting approach can be taken. It is simply waiting for running tasks that should be preempted to finish, meaning that not taking any action for achieving the several goals as we described above. Waiting can be efficient when a cluster runs many short jobs. However, it can introduce a considerable amount of delay for the preempting jobs in case of long-running tasks that have to be preempted.

Besides wait and kill approaches, we can also apply work-conserving preemption technique to achieve our goals. Interestingly, this technique is neither supported by Hadoop nor has been investigated by the scientific community in detail. For clarity, we illustrate these three techniques (i.e., wait, kill and preemption) in Figure 6.3. It presents a scenario where a shorter task, *t2*, needs a slot from the longer one as it arrives in the middle of the execution of the longer task, *t1*. With the wait approach, we can see that the execution time of *t2* prolonged as much as the remaining time of *t1* until its completion. With the kill approach, we observe the longest execution time where on-going work of *t1* is wasted and needs to be re-executed after the completion of *t2*. In the last scenario with the work-conserving preemption approach, we can see that all the work that has been done by the longer task has been conserved at the moment of the preemption. After *t2* finishes its execution, *t1* continues its execution where it left off. Here, Δt represents the overhead that can be caused by the preemption technique. This scenario promises the lowest average waiting time for both jobs which motivated us to leverage it within Chronos. In this work, Chronos leverages it for better handling of failures by pausing the running tasks to make room for the recovery tasks, and resuming them from their paused state when there is an available slot.

For the preemption mechanism, a naive checkpointing approach [111] can be taken as a strawman solution. Although this approach is simple to implement, it will introduce a large overhead since checkpointing requires flushing all the data associated with the preempted task. Moreover, obtaining this checkpoint upon resume can consume a significant amount of network and I/O resources. In this section, we introduce our lightweight preemption technique for map and reduce task preemption.

Map Task Preemption

During the map phase, TaskTracker executes the map tasks that are assigned to it by the JobTracker. Each map task processes a chunk of data (input block) by looping through every key-value pair and applying the user defined map function. When all the key-value pairs have been processed, JobTracker will be notified as the map task is completed and the intermediate map output will be stored locally to serve the reducers.

For the map task preemption, we introduce an *earlyEnd* action for map tasks. We also augment each map task with its range information which includes the starting and the ending key for the key-value pairs. The map task listens for the preemption signal from Chronos in order to stop at any time. Upon receiving the preemption request, this action will stop the looping procedure and split the current map task into two subtasks. The former subtask covers all the key-value pairs that have been processed before the preemption request comes. This subtask will be reported back to the JobTracker as completed as in the normal map task

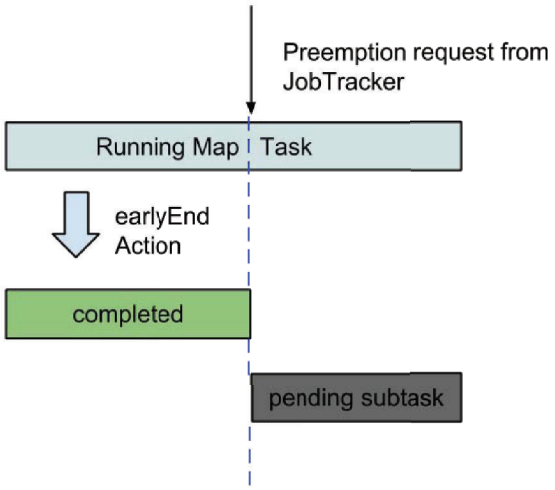


Figure 6.4: Map task preemption.

execution. On the other hand, the second subtask contains the range information for the key-value pairs that have not been processed yet. This subtask will be added to the map pool for later execution when there is an available slot, as for new map tasks. This map task preemption mechanism is illustrated in Figure 6.4. Full parallelism of map tasks by having independent key-value pairs gives us the opportunity to have fast and lightweight map preemption.

To ensure the correctness of our map preemption mechanism, we also notify the reduce tasks for the new subtasks created upon the map task preemption. In Hadoop, Job Tracker notifies reduce tasks about the location of intermediate map outputs through Map Completion Event message. Reduce task would assume correct execution if it receives the same number of *Map Completion Event* messages as the number of map tasks which was initialized in the beginning of the job. Since our mechanism introduces new subtasks upon preemption, we include this information inside the Map Completion Event message to inform the reduce tasks for these new subtasks. Next, we present our reduce task preemption mechanism.

Reduce Task Preemption

In Hadoop, reduce task execution consists of three phases: shuffle, sort and reduce. During the shuffle phase, reducers obtain the intermediate map outputs for the key-set assigned to them. The sort phase performs a sort operation on all the fetched data (i.e., intermediate map outputs in the form of key-value pairs). Later, the reduce phase produces the final output of the MapReduce job by applying the user defined reduce function on these key-value pairs.

For the reduce preemption, the splitting approach (as in the map preemption) would not be feasible due to the different characteristics of map and reduce tasks. Full parallelism of map execution and having map inputs on the distributed file system enables us to apply a splitting approach for map preemption. However, the three different phases of the reducer

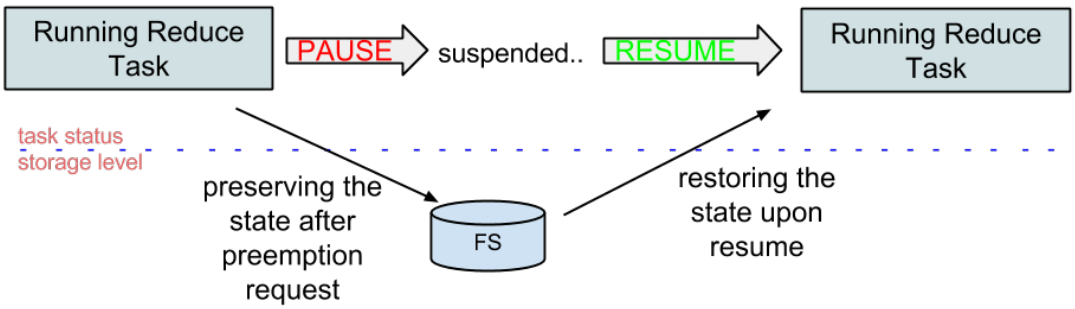


Figure 6.5: Reduce task preemption.

are not fully independent of each other. Therefore, we opt for a pause-and-resume approach for the reduce preemption. In brief, we store the necessary data on the local storage for preserving the state of the reduce task with pause and we restore back this information upon resume. Figure 6.5 displays this reduce task preemption mechanism.

Our reduce preemption mechanism can preempt a reduce task at any time during the shuffle phase and at the boundary of other phases. The reason behind this choice is that usually the shuffle phase covers a big part of the reduce task execution, while the sort and reduce phases are much shorter. In particular, the sort phase is usually very short due to the fact that Hadoop launches a separate thread to merge the data as soon as it becomes available.

During the shuffle phase, reducers obtain the intermediate map outputs for the key-set assigned to them. Then, these intermediate results are stored either on the memory or local disk depending on the memory capacity of the node and also the size of the fetched intermediate results [65]. Upon receiving a preemption request, the pause action takes place and first stops the threads that fetch the intermediate results by allowing them to finish the last unit of work (i.e., one segment of the intermediate map output). Then, it stores all the in-memory data (i.e., number of copied segments, number of sorted segments) to local disk. This information is kept in files that are stored in each task attempt's specific folder, which can be accessed later by the resumed reduce task.

Preemption at the boundary of the phases follows the same procedure as above. The data necessary to preserve the state of the reduce task is stored on the local disk and then the reduce task will release the slot by preempting itself. The task notifies the JobTracker with a status of *suspended*. Suspended tasks will be added to the reduce pool for later execution when there is an available slot.

6.3 Experimental Evaluation

In this section, we evaluate Chronos by comparing it with the state-of-the-art Hadoop schedulers with representative Big Data workloads using Grid'5000 [19] testbed.

6.3.1 Methodology

Platform

We run our experiments on the Rennes site of Grid'5000 testbed. Specifically, we use the parapluie cluster. Each node of this cluster is outfitted with *12-core* AMD *1.7 GHz* CPUs and *48 GB* of RAM. Intra-cluster communication is done through a *1 Gbps* Ethernet network.

Hadoop deployment

We configured and deployed a Hadoop cluster using *9 nodes*. The Hadoop instance consists of the NameNode and the JobTracker, both deployed on a dedicated machine, leaving *8 nodes* to serve as both DataNodes and TaskTrackers. The TaskTrackers were configured with *8 slots* for running map tasks and *4 slots* for executing reduce tasks. At the level of HDFS, we used a chunk size of *256 MB* due to the large memory size in our testbed. We set a replication factor of *2* for the input and output data. As suggested in several studies in the literature [32], we set the failure detection timeout to a smaller value (i.e., *25 seconds*) compared to the default timeout of *600 seconds*, since the default timeout is too big compared to the likely completion time of our workloads in failure-free periods.

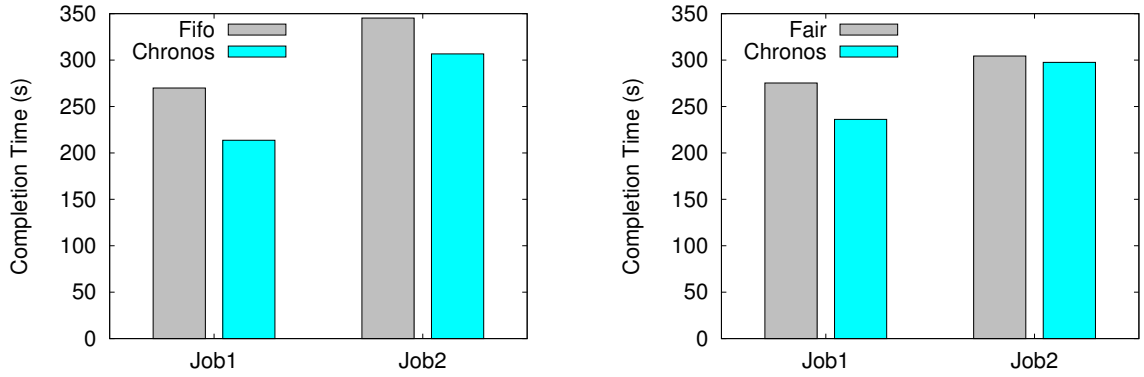
Failure injection: To mimic the failures, we simply killed the TaskTracker and DataNode processes of a random node. We could only inject one machine failure since Hadoop cannot tolerate more failures due to the replication factor of *2* for HDFS.

Workloads

We evaluated Chronos using two representative Big Data workloads (i.e., wordcount and sort) with different input data sizes from the PUMA datasets [4]. Wordcount is a Map-heavy workload with a light reduce phase, which accounts for about 70% of the jobs in Facebook clusters [26]. On the other hand, sort produces a large amount of intermediate data which leads to a heavy reduce phase, therefore representing Reduce-heavy workloads, which accounts for about 30% of the jobs in Facebook clusters [26].

Comparisons

We implemented Chronos in Hadoop-1.2.1 with two state-of-the-art Hadoop schedulers: Fifo (i.e., priority scheduler with respect to job submission time) and Fair schedulers. We compare Chronos to these baselines. The Fifo scheduler is the default scheduler in Hadoop and is widely used by many companies due to its simplicity, especially when the performance of the jobs is the main goal. On the other hand, the Fair scheduler is designed to provide fair allocation of resources between different users of a Hadoop cluster. Due to the increasing numbers of shared Hadoop clusters, the Fair scheduler also has been exploited recently by many companies [70, 150].



(a) Chronos *vs* Fifo scheduler.

(b) Chronos *vs* Fair scheduler.

Figure 6.6: Performance comparison for Map-Heavy jobs.

6.3.2 The Effectiveness of Chronos in Reducing Job Completion Times

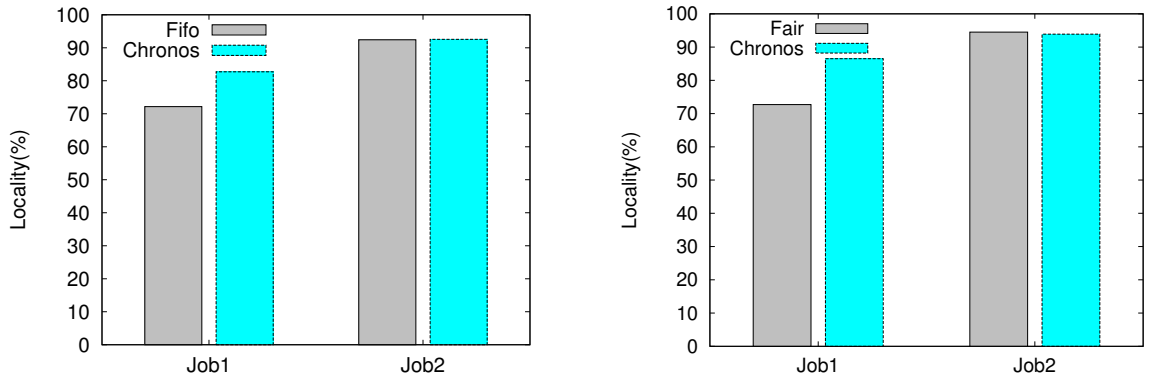
Results with Fifo Scheduler

We ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. The input data sizes result in fairly long execution times of both jobs, which allowed us to thoroughly monitor how both Hadoop and Chronos handles machine failures. More importantly, this mimics a very common scenario when small and long jobs concurrently share a Hadoop cluster [108, 150]. Also, we tried to ensure that the cluster capacity (64 map slots in our experiments) is completely filled. After submitting the jobs, we have injected the failure before the reduce phase starts in order to have only map tasks as failed tasks.

In contrast to Fifo, Chronos reduces the completion time of the first job and the second one by 20% and 10%, respectively. Most of the failed tasks belong to the first job and therefore Chronos achieves better performance for the first job compared to the second one. The main reason for the performance improvement is the fact that Fifo waits until there is a free slot before launching the recovery tasks, while Chronos launches recovery tasks shortly after failure detection. The waiting time for recovery tasks is *51 seconds* (15% of the total job execution) in the Fifo scheduler and only *1.5 seconds* in Chronos (Chronos waited *1.5 seconds* until new heartbeats arrived). Moreover, during this waiting time, recovery tasks from the first submitted job (high priority job) are waiting while tasks belonging to the second job (low priority) are running tasks. This obviously violates the Fifo scheduler rule. Therefore, the significant reduction in the waiting time not only improves the performance but also ensures that Fifo operates correctly towards its objective.

Results with Fair Scheduler

We ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. The failure is injected before the reduce phase starts. Figure 6.6(b) demonstrates that Chronos improves the job completion time by 2% to 14%, compared to Fair scheduler. This behavior stems from eliminating the waiting time for recovery tasks besides launching them locally. We observe that failure results in a serious fairness problem between jobs with



(a) Data locality in Chronos and Fifo scheduler.

(b) Data locality in Chronos and Fair scheduler.

Figure 6.7: Data locality for Map-Heavy jobs under Chronos, Fifo and Fair Schedulers.

Hadoop's Fair scheduler: this fairness problem (violation) lasts for almost *48 seconds* (16% of the total execution time) in the Fair scheduler, while Chronos restores fairness within about *2 seconds* by preempting the tasks from the jobs which exceed their fair share.

Summary. One may think that preempting tasks from low priority jobs to launch recovery tasks of high priority jobs will obviously result in a performance degradation for low priority jobs in Chronos compared to both the Fifo and Fair schedulers. However, the performance improvements of high priority jobs in Chronos (due to the waiting time reduction and data locality improvement) result in an earlier release of slots and therefore tasks belonging to low priority jobs are launched earlier and fewer tasks are competing with them for resources.

6.3.3 The Effectiveness of Chronos in Improving Data Locality

Besides the failure handling, Chronos also aims at mitigating the I/O latency by trying to launch recovery tasks locally. Figure 6.7 shows the data locality of each job from previous experiments with Fifo (Figure 6.7(a)) and Fair (Figure 6.7(b)) schedulers. While the second job has a similar data locality, we can clearly observe that Chronos significantly improves the data locality for the first job for both scenarios (i.e., 15% and 22% data locality improvement compared to Fifo and Fair schedulers, respectively). This improvement is due to the almost optimal locality for recovery tasks with Chronos (all the recovery tasks which are launched through Chronos are executed locally). Only 12.5% of the recovery tasks were executed locally in Hadoop. The improved locality brings better resource utilization by eliminating the need for remote transfer of input blocks for recovery tasks and further improves the performance.

Summary. The aforementioned results demonstrate the effectiveness of Chronos in reducing the violation time of the scheduler (i.e., priority based on job submission time and fairness) to a couple of seconds. More importantly, Chronos reduces the completion time of the first job (the job was affected by the machine failure) due to the reduction in the waiting time and optimal locality for recovery tasks. This in turn allows the second job to utilize all available resources of the cluster and therefore improves the performance.

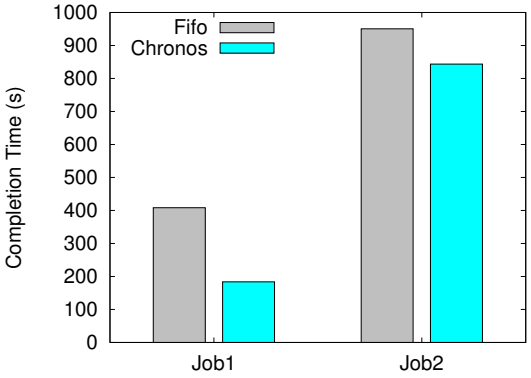


Figure 6.8: Job completion times for Reduce-Heavy jobs under Chronos and Fifo scheduler.

6.3.4 Impact of Reduce-Heavy Workloads

We also evaluated Chronos with Reduce-Heavy workloads. We ran two sort applications with input data sizes of *17 GB* and *56 GB*, respectively. Both jobs have *32 reduce* tasks. We injected the failure during the reduce phase in order to have failed reduce tasks from the first job. Figure 6.8 details the job completion time with Chronos and Fifo. Chronos achieves a 55% performance improvement for the first job and 11% for the second one. The improvement in the Reduce-Heavy benchmark is higher compared to the Map-Heavy benchmark because reduce tasks take a longer time until they are completed and therefore the recovery (reduce) tasks have to wait almost *325 seconds* in Fifo. Chronos successfully launches recovery tasks within *2 seconds*.

Summary. Chronos achieves a higher improvement when the failure injection is in the reduce phase which clearly states that the main improvement is due to the reduction in the waiting time. Here it is important to mention that other running reduce tasks will be also affected by the waiting time as they need to re-fetch the lost map outputs (produced by the completed map tasks on the failed machine).

6.3.5 The Effectiveness of the Preemption Technique

To assess the impact of the preemption technique on Chronos performance, we have also implemented Chronos with a kill primitive as a preemption technique (Chronos-Kill). Instead of pausing the candidate tasks to free the slot, we kill the candidate tasks with Chronos-Kill. We repeated the same experiments as above, and Figure 6.9(a) shows the results. Although both implementations have similar completion times for first job, Chronos-Kill degrades the completion time of the second job by 12.5%. Note that with Fifo scheduler, the first job has a higher priority compared to the second job due to its earlier submission time. Chronos-Kill thus kills the tasks from the second job to allocate slots for recovery tasks. This results in a waste of resources (we have observed that more than 50% of the killed tasks are killed after *60 seconds* of execution, as shown in Figure 6.9(b)). On the other hand, our preemption mechanism has a work-conserving behavior in which the preempted tasks from the second job continue their execution without wasting the work that has already been done.

6.3.6 Overhead of Chronos

The overhead of Chronos may be caused by two factors: first, due to the collection of the useful information (i.e., real-time progress reports) that is fed later to our smart slot allocation strategy, and second, due to the overhead of the preemption technique. With respect to the slot allocation strategy, the overhead of Chronos is negligible because Chronos leverages the information already provided by heartbeat messages. We have studied the overhead of the preemption technique by repeating the same experiment as in Section 6.3.2. Figure 6.10(a) shows the completion times of each successful task with Chronos and Hadoop, we can see that they both have a similar trend. Thus, we conclude that the preemption technique does not add any noticeable overhead to cluster performance in general.

Moreover, we studied the overhead of Chronos during the normal operation of the Hadoop cluster. We ran the same experiment as in Section 6.3.2 five times without any failures and Figure 6.10(b) shows that Chronos incurs negligible performance overhead during the normal operation.

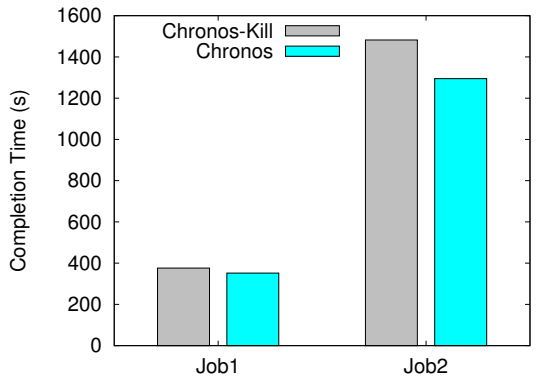
6.3.7 Chronos and Hadoop Under Multiple Failures

Our results demonstrated that the performance of Big Data applications degrade significantly with a single failure. Furthermore, it is common to have multiple failures in clouds since single failures can have a cascading effect [29, 32, 109, 135]. For instance, Rosa et al. analyzed the traces from Google production cluster and found out that more than 55% of failed tasks experience multiple failures during their lifetime [109]. Hence, we also evaluated Chronos under multiple failures. We repeated the same Hadoop deployment in previous experiments by only changing the replication factor of HDFS to 3 in order to tolerate two failures.

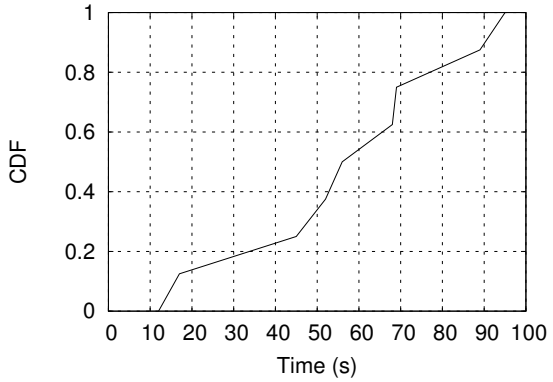
We ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. After submitting the jobs, we have injected two failures before the reduce phase starts in order to have only map tasks as failed tasks. Figure 6.11 illustrates that Chronos again reduces the completion time of the jobs by 22% and 13% thanks to its early yet smart slot allocation strategy for recovery tasks.

6.3.8 Chronos with Aggressive Slot Allocation

The main objective of Chronos is to correct the behavior of Hadoop schedulers after failure. Therefore, once this is achieved, Chronos operates similarly to Hadoop's original mechanism to avoid preempting tasks from the same job. However, after observing that Chronos's preemption technique has a negligible overhead, we wanted to assess the impact of agressive slot allocation. To this end, we allow Chronos to preempt the selected tasks with the same priority for the recovery tasks (e.g., recovery tasks belonging to the same job with selected tasks). Previously, recovery tasks with higher priority would preempt the selected tasks with less priority in Chronos. One may think that preempting tasks from the same job to launch recovery tasks would not improve the job performance. However, thanks to the work-conserving preemption technique we can safely preempt the tasks even belonging to the same job to improve the locality of recovery tasks. Here, we name Chronos with aggressive slot allocation as Chronos* for brevity.

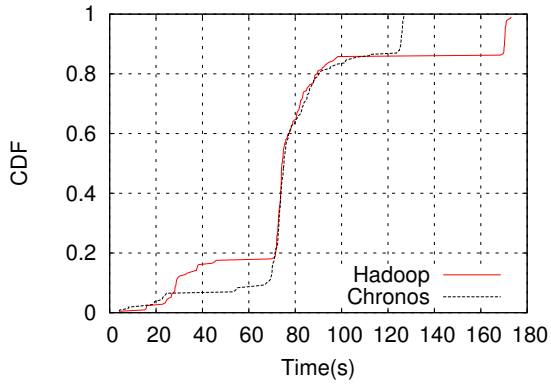


(a) Job completion times.

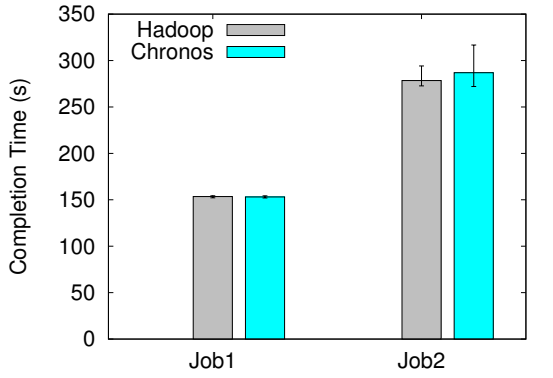


(b) CDF of running times of killed tasks, including map and reduce tasks.

Figure 6.9: Job completion times for Reduce-Heavy jobs under Chronos and Chronos-Kill.



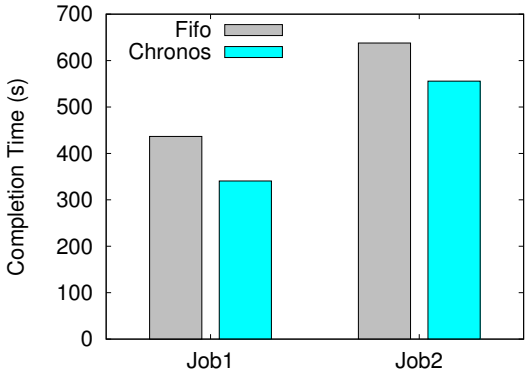
(a) CDFs of completion times of successful tasks under Chronos and Fifo scheduler.



(b) Overhead of Chronos during normal operation.

Figure 6.10: Overhead of Chronos.

Figure 6.11: Performance comparison of Big Data jobs under double failure.



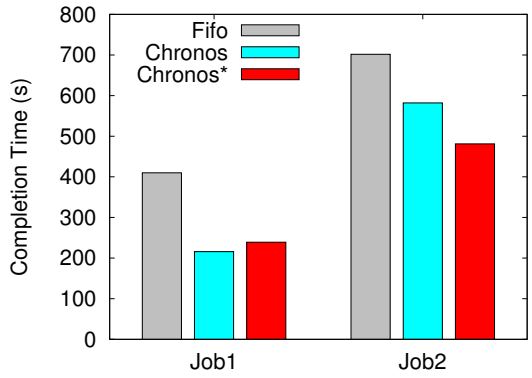


Figure 6.12: Job completion times under Fifo, Chronos and Chronos* schedulers with single failure.

To assess the effectiveness of Chronos*, we ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. The failure is injected before the reduce phase starts. We adjusted the network speed to *256 Mbps* in order to incorporate the locality factor on the performance. The results are shown in Figure 6.12. Although both implementations have similar completion times for the first job, Chronos* reduces the completion time of the second job by 17% compared to Chronos. Note that with Chronos, recovery tasks that belong to the second job would not be able to preempt any of the running tasks due to their later submission time. However, we observed 100% locality for recovery tasks with Chronos* thanks to its aggressive slot allocation strategy by also allowing the recovery tasks to preempt the tasks from the same job. This highlights the importance of mitigating I/O latencies in shared Hadoop clusters besides efficiently handling failures.

6.3.9 Discussion of the Results

Fail-stop failures can severely decrease the performance of Big Data applications [32, 66, 107]. Our results demonstrate that Chronos recovers to a correct scheduling behavior within a couple of seconds only and reduces the job completion times compared to the state-of-the-art schedulers. The performance improvement is due to two main factors:

- *Reduction in the waiting time:* The reduction in waiting time varies according to the status and the progress speed of running tasks when detecting the failure. However, as shown in many studies, recently Hadoop cluster is shared by multiple different Big Data applications (i.e., differ in their complexity and thus in the average execution time of their tasks [108, 150]: by analyzing the traces collected from three different research clusters [52], we observe that the average execution time of map and reduce tasks is 124 *seconds* and 901 *seconds*, respectively). Thus, having long running tasks, despite the failure injection and detection times, is common in Hadoop clusters. As a result, there is a significant potential for improving the performance of jobs using Chronos.

- *Improved locality of recovery tasks:* Network is normally the most scarce resource in today's clouds [115]. Chronos therefore, by launching local recovery tasks, reduces the extra cost for data transferring of these sensitive tasks and improves the performance of Big Data applications.

It is also important to mention that although we implemented Chronos in Hadoop 1.2.1, the same logic can also be applied to the next generation of Hadoop, YARN [132]. Major difference of YARN from the Hadoop 1.2.x versions is that it separates the JobTracker into ResourceManager and ApplicationManager. The main motivation behind this new design is to provide better fault tolerance and scalability. However, our initial experiences with YARN indicate that severe impact of failures still exists since its schedulers are also unaware of failures and YARN adopts the same fault tolerance mechanism as Hadoop 1.2.x.

6.4 Related Work

6.4.1 Scheduling in Big Data Systems

There exists a large body of studies on exploring new objectives (e.g., fairness, job priority) when scheduling multiple jobs in Big Data systems and improving their performance. Isard et al. introduced Quincy [70], a fair scheduler for Dryad, which treats scheduling as an optimization problem and uses min-cost flow algorithm to achieve the solution. Quincy uses the kill mechanism to set the cluster according to the configuration in the solution. Zaharia et al. introduced a delay scheduler [150], a simple delay algorithm on top of the default Hadoop Fair scheduler. Delay scheduling leverages the fact that the majority of the jobs in production clusters are short, therefore when a scheduled job is not able to launch a local task, it can wait for some time until it finds the chance to be launched locally. More recently, Venkataraman et al. have proposed KMN [134], a scheduler that focuses on applications with input choices and exploits these choices for performing data-aware scheduling. KMN also introduces additional map tasks to create choices for reduce tasks. This in turn results in less congested links and better performance. Although these scheduling policies can improve the performance of Big Data applications, none of them is failure-aware, leaving the fault tolerance mechanism to the Big Data system itself, and thus are vulnerable to incurring uncertain performance degradations in case of failures. Moreover, Chronos can complement these policies to enforce correct operation and to further improve their performance under failures.

6.4.2 Exploiting Task Preemption in Big Data Systems

There have been a few studies on introducing work-conserving preemption techniques to Big Data environments. Wang et al. [140] exploited the fact that long running reduce tasks may lead to starvation of short jobs. Thus, they introduced a technique for reduce task preemption in order to favor short jobs against long jobs. However, the preemption technique is limited to reduce tasks and there is no overhead study regarding the preemption technique. Pastorelli et al. [101] have proposed a preemption technique with a pause and resume mechanism to enforce the job priority levels for the job execution. For the pause and resume mechanism, they leverage the already available memory management mechanisms in the operating system. Although these mechanisms can simplify the preemption, it brings the shortcoming that a suspended task can only be launched on the same machine on which it was paused before. Ananthanarayanan et al. [7] introduced Amoeba to support a lightweight checkpointing mechanism for reduce tasks with the aim of achieving better

elasticity for resource allocation. In contrast, Chronos introduces both map and reduce task preemption and leverages it to make Hadoop schedulers failure-aware.

6.5 Conclusions

Hadoop has emerged as a prominent tool for Big Data processing in large-scale clusters. Failures are inevitable in these clusters, especially in shared environments. Consequently, Hadoop was designed with hardware failures in mind. In particular, Hadoop handles machine failures by re-executing all the tasks of the failed machine. Unfortunately, the efforts to handle failures are entirely entrusted to the core of Hadoop and hidden from Hadoop schedulers. This may prevent Hadoop schedulers from meeting their objectives (e.g., fairness, job priority, performance) and can significantly impact the performance of the applications.

Given the limitations of current Hadoop schedulers in handling the failures, we propose a new scheduling strategy called Chronos. Chronos is conductive to improving the performance of Big Data applications by enabling an early action upon failure detection. Chronos tries to launch recovery tasks immediately by preempting tasks belonging to low priority jobs, thus avoiding the uncertain time until slots are freed. Moreover, Chronos strongly considers the local execution of recovery tasks. The experimental results indicate that Chronos results in almost optimal locality execution of recovery tasks and improves the overall performance of Big Data applications by up to 55%. Chronos achieves that while introducing very little overhead.

Chapter 7

Conclusions and Perspectives

Contents

7.1 Achievements **98**
7.2 Perspectives **99**

As data volumes growing at a dramatic rate, Big Data applications has become crucial in our lives by extracting meaningful information from this data. A key issue is to run these applications efficiently on large-scale shared platforms to be able to process these gigantic data volumes. Several factors play an important role in the efficiency of these applications in terms of their performance:

1. Data movements within the data processing platforms affect the application performance dramatically. With larger amounts of data being generated everyday, it becomes a must to develop solutions that can reduce the amount of these data movements.
2. I/O interference is a major performance bottleneck due to the shared nature of the data processing platforms. This interference problem will be more important as these platforms are being used by more and more concurrent applications. Hence, smart I/O interference mitigation strategies are necessary to prevent it from being bottleneck to the performance of Big Data applications.
3. Finally, failures can significantly degrade the performance of Big Data applications. Failures are already part of the data processing platforms and they are becoming more severe as these platforms are getting larger and larger.

In this thesis, we addressed aforementioned issues to achieve efficient Big Data processing on large-scale shared platforms through a number of contributions that we describe next. Then, we discuss the perspectives that our research opens for Big Data processing on large-scale shared platforms.

7.1 Achievements

The achievements obtained in this thesis can be summarized as follows.

Investigating the Characteristics of Big Data Applications in HPC Systems

As a first step toward efficient Big Data processing on HPC systems, we conducted an experimental campaign to characterize the performance of Big Data applications on these systems. We ran Spark using representative Big Data workloads on Grid'5000 testbed to evaluate how the latency, contention and file system's configuration can influence the performance of Big Data applications. Our study demonstrated that there are several performance issues when running Spark on a HPC system. For instance, we observe that I/O latency resulting from the data movements between the parallel file system and compute nodes can significantly degrade the application performance. Moreover, we report that this latency problem is significant for all I/O phases, in contrary to existing studies in the literature. Our findings also illustrate that one should carefully tackle the I/O interference problem, mainly resulting from sharing the same storage system, when running Big Data applications on a HPC system.

Given all these performance issues, we claim that blind adoption of Spark or other mainstream data processing frameworks can not leverage HPC systems efficiently for Big Data processing. This rises the need for novel I/O management solutions that can alleviate the aforementioned performance bottlenecks and enable efficient Big Data processing on these systems.

Studying I/O Interference in HPC Systems

Performance characteristics of Big Data applications highlighted I/O interference as one of the major bottlenecks. With larger machines, more and more applications would run concurrently. Hence, tackling this interference problem will play a more important role in enabling efficient Big Data processing on HPC systems. To this end, we conduct an extensive experimental campaign to understand the root causes of I/O interference. We use microbenchmarks on the Grid'5000 to evaluate how the applications' access pattern, the network components, the file system's configuration, and the backend storage devices influence interference. Our experiments highlight the different causes of I/O interference in several scenarios and are accompanied by lessons learned from the behavior of the system in these scenarios. An important outcome of our study is that in many situations, interference is a result of bad flow control in the I/O path, rather than the presence of a single bottleneck in one of its components. Hence, we believe that researchers must understand the tradeoffs between several components in the I/O stack and must address the interference problem in its entirety, rather than focusing on any single component. We hope that the insights and lessons learned from our experiments will enable a better understanding of I/O interference and help enable efficient Big Data processing on HPC systems.

Extending Burst Buffers in HPC Systems to Enable Efficient Big Data Processing

I/O management solutions play a critical role in achieving efficient Big Data processing on HPC systems. To this end, we proposed the Eley approach. Eley is a burst buffer solution that precisely aims to reduce the I/O latencies when reading the input data. Eley also controls the I/O interference that can be caused by these read requests of Big Data applications on HPC applications that are sharing the same platform. Eley employs interference-aware prefetching technique to improve the performance of Big Data applications while guaranteeing the QoS of HPC applications. Our extensive evaluation with both real system experiments on Grid'5000 and simulations demonstrated that Eley reduces the execution time of Big Data applications by up to 30% compared to the existing burst buffer solutions while guaranteeing the QoS requirement of HPC applications.

Addressing Failures in Large Shared Clusters

Besides I/O management, failure handling also plays a crucial role in performing efficient Big Data processing on large-scale shared platforms, given their scale. With these platforms getting larger and larger, it becomes necessary to have efficient failure recovery mechanisms. To this end, we propose the Chronos approach. Chronos is a failure-aware scheduler that enables an early yet smart action for fast failure recovery. To achieve this, Chronos uses a novel preemption technique and preempts the tasks belonging to low priority jobs to launch the recovery tasks immediately. Furthermore, Chronos aims at sustaining a high data locality even under failures to avoid the extra data transfers that can be necessary to ship the required data to recovery tasks. Our experiments on Grid'5000 demonstrated that Chronos mitigates the impact of failures on the performance of Big Data applications while introducing a negligible overhead.

7.2 Perspectives

Our present work addresses several issues related to the I/O management and failure handling at large-scale shared platforms with the aim of achieving efficient Big Data processing on these platforms. Towards this goal, our conducted research naturally opens a number of perspectives.

Developing an I/O Interference Simulator

Interference is an inevitable fact that large-scale systems face when multiple applications (i.e., Big Data and HPC applications) share the same platform. In Chapter 4, we undertook an effort to identify the root causes of I/O interference in HPC storage systems and demonstrated that interference results from the interplay between several components in the I/O stack. An immediate question that can be asked is that can we design a simulator which can accurately model the components subject to this interference. In this direction, we first plan to expand our experimental study by looking at other platforms than Grid'5000, other file systems (e.g., Lustre), other workload types (e.g., read-only, mixed) and other types of network (e.g., InfiniBand). Then, by leveraging the knowledge gained in our work, we plan to design this event-driven I/O interference simulator.

Leveraging Eley for the Output Phase

Eley constitutes a good step toward developing smart I/O management solutions to enable efficient Big Data processing on HPC systems. Currently, Eley can reduce the latencies for the input and intermediate data phases of Big Data applications. Therefore, an important research direction in the near future will be to leverage Eley for the output phase as well. This will alleviate the latency problem in all I/O phases. Moreover, Chapter 4 demonstrated that write/write interference can be very severe for the application performance. Therefore, we plan to extend the interference-aware data transfer strategies so that synchronizing the output data to the parallel file system will not violate the performance requirements of HPC applications.

Equipping Eley with Smart Data Eviction Policies

Eley improves the performance of Big Data applications by allowing them to perform high-throughput I/O operations. However, there is a trade-off between throughput and storage capacity when using burst buffers. In Chapter 3, we highlighted that the limited storage capacity of burst buffers can have a significant impact on the performance of Big Data applications. To address this challenge, Eley prefetches a subset of the input data (i.e., one wave) as compute cluster computes one wave at a time. After extending Eley for reducing the latencies in the output phase, one concern would be how to partition the storage space of burst buffers with respect to different I/O phases (i.e., input, intermediate and output) of Big Data applications. To this end, smart data eviction policies will be needed. These policies can minimize the I/O operations between compute nodes and parallel file system by trying to store the useful data for Big Data applications on burst buffers.

Extending Chronos to HPC Systems

Since failures are more severe in clouds due to employing failure-prone commodity machines, we focus on the failure handling in clouds by employing Chronos in shared Hadoop clusters in this thesis. Chronos demonstrated its effectivess in improving the performance of Big Data applications under failures. On the other hand, failures are becoming more common on HPC systems due to the ever-increasing scale of these systems. For example, the BlueWaters supercomputer at the National Center of Supercomputing [18] has a mean time to failure of 4.2 hours [31]. One potential research perspective would thus consists of leveraging Chronos to mitigate the impact of failures on the performance of Big Data applications in HPC systems. However, this is not a trivial task considering the different architectural designs in clouds and HPC systems. As shown in Chapter 6, improved locality of recovery tasks is one of the main factors for the performance improvement brought by Chronos besides the reduction in the waiting time for these tasks. Since all the compute nodes have an equal distance to the data storage (i.e., parallel file system) in HPC systems, we would need new methods to integrate this factor for Chronos when extending it to HPC systems. For instance, it would be interesting to investigate the feasability of employing Eley to fetch the input data for recovery tasks and thus reducing the I/O latency when launching the recovery tasks.

Bibliography

[1] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: scalable data staging services for petascale applications", in *International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2009, pp. 39–48.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: queries with bounded errors and bounded response times on very large data", in *European Conference on Computer Systems*, ACM, 2013, pp. 29–42.

[3] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "ShuffleWatcher: shuffle-aware scheduling in multi-tenant MapReduce clusters.", in *Annual Technical Conference*, USENIX, 2014, pp. 1–12.

[4] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue MapReduce benchmarks suite", Purdue University, Tech. Rep., 2012.

[5] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems", in *International Conference on Cluster Computing and Workshops*, IEEE, 2009, pp. 1–10.

[6] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in MapReduce clusters", in *European Conference on Computer Systems*, ACM, 2011, pp. 287–300.

[7] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica, "True elasticity in multi-tenant data-intensive compute clusters", in *Symposium on Cloud Computing*, ACM, 2012, pp. 1–7.

[8] *Apache Apex*, 2016. [Online]. Available: https://apex.apache.org/ (visited on 08/08/2017).

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing", *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark SQL: relational data processing in Spark", in *International Conference on Management of Data*, ACM, 2015, pp. 1383–1394.

[11] G. Aupy, A. Gainaru, and V. L. Fèvre, "Periodic I/O scheduling for supercomputers", 2017.

[12] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: an introduction to the design of warehouse-scale machines", in *Synthesis Lectures on Computer Architecture*, 3, vol. 8, Morgan & Claypool Publishers, 2013, pp. 1–154.

[13] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey, "CA-NFS: a congestion-aware network file system", *Transactions on Storage*, vol. 5, no. 4, pp. 1–24, 2009.

[14] *Apache Beam*, 2016. [Online]. Available: https://beam.apache.org/ (visited on 08/08/2017).

[15] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-free co-processing on a prototype exascale storage stack", in *International Symposium on Mass Storage Systems and Technologies*, IEEE, 2012, pp. 1–5.

[16] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2013, pp. 1–12.

[17] *Big Data and Extreme-scale computing workshop*, 2013. [Online]. Available: http://www.exascale.org/bdec/ (visited on 07/30/2017).

[18] *BlueWaters project*, National Center for Supercomputing Applications, 2010. [Online]. Available: http://www.ncsa.illinois.edu/BlueWaters/ (visited on 08/04/2017).

[19] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed", *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[20] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters", *International Journal on Very Large Databases*, vol. 3, no. 1-2, pp. 285–296, 2010.

[21] P. Buneman and W.-C. Tan, "Provenance in databases", in *International Conference on Management of data*, ACM, 2007, pp. 1171–1173.

[22] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility", *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[23] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: stream and batch processing in a single engine", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, pp. 28–38, 2015.

[24] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads", in *International Conference on Cluster Computing*, IEEE, 2009, pp. 1–10.

[25] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling Spark on HPC systems", in *International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2016, pp. 97–110.

[26] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in Big Data systems: a cross-industry study of MapReduce workloads", *International Journal on Very Large Databases*, vol. 5, no. 12, pp. 1802–1813, 2012.

[27] N. Cheriere, P. Donat-Bouillud, S. Ibrahim, and M. Simonin, "On the usability of shortest remaining time first policy in shared Hadoop clusters", in *Annual Symposium on Applied Computing*, ACM, 2016, pp. 426–431.

[28] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: design and evaluation of eviction policies for supporting priorities and deadlines in MapReduce clusters", in *Annual Symposium on Cloud Computing*, ACM, 2013, pp. 1–17.

[29] J. Dean, "Large-scale distributed systems at Google: current systems and future directions", in *International Workshop on Large Scale Distributed Systems and Middleware*, Tutorial, ACM, 2009.

[30] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[31] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: the case of BlueWaters", in *International Conference on Dependable Systems and Networks*, IEEE, 2014, pp. 610–621.

[32] F. Dinu and T. Ng, "Understanding the effects and implications of compute node related failures in Hadoop", in *International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2012, pp. 187–198.

[33] F. Dinu and T. E. Ng, "RCMP: enabling efficient recomputation based failure resilience for Big Data analytics", in *International Parallel and Distributed Processing Symposium*, IEEE, 2014, pp. 962–971.

[34] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems", Sandia Report, SAND2013-4744, Tech. Rep., 2013, pp. 1–18.

[35] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: how to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O", in *International Conference on Cluster Computing*, IEEE, 2012, pp. 155–163.

[36] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf, "Damaris: addressing performance variability in data management for post-petascale simulations", *Transactions on Parallel Computing*, vol. 3, no. 3, pp. 1–43, 2016.

[37] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: mitigating I/O interference in HPC systems through cross-application coordination", in *International Parallel and Distributed Processing Symposium*, IEEE, 2014, pp. 155–164.

[38] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2014, pp. 623–634.

[39] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for data intensive scientific analyses", in *International Conference on eScience*, IEEE, 2008, pp. 277–284.

[40] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan, "Mariane: MapReduce implementation adapted for HPC environments", in *International Conference on Cluster, Cloud and Grid Computing*, IEEE, 2011, pp. 82–89.

[41] J. Flich, G. Agosta, P. Ampletzer, D. A. Alonso, C. Brandolese, A. Cilardo, W. Fornaciari, Y. Hoornenborg, M. Kovač, B. Maitre, *et al.*, "Enabling HPC for QoS-sensitive applications: the MANGO approach", in *Design, Automation and Test in Europe Conference and Exhibition*, IEEE, 2016, pp. 702–707.

[42] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "Big Data, simulations and HPC convergence", in *Workshop on Big Data Benchmarks*, Springer, 2015, pp. 3–17.

[43] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC applications under congestion", in *International Parallel and Distributed Processing Symposium*, IEEE, 2015, pp. 1013–1022.

[44] S. K. Garg, C. S. Yeo, A. Anandasivam, and R. Buyya, "Environment-conscious scheduling of HPC applications on distributed cloud-oriented data centers", *Journal of Parallel and Distributed Computing*, vol. 71, no. 6, pp. 732–749, 2011.

[45] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: fair allocation of multiple resource types", in *International Symposium on Networked Systems Design and Implementation*, USENIX, vol. 11, 2011, pp. 323–336.

[46] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: max-min fair sharing for datacenter jobs with constraints", in *European Conference on Computer Systems*, ACM, 2013, pp. 365–378.

[47] *Apache Giraph*, 2013. [Online]. Available: https://giraph.apache.org/ (visited on 08/08/2017).

[48] *What happens on Internet every 60 seconds*, Go-Globe, 2016. [Online]. Available: http://www.go-globe.com/blog/60-seconds/ (visited on 06/27/2017).

[49] *Google Cloud Platform*, Google, 2017. [Online]. Available: https://cloud.google.com/compute/ (visited on 06/26/2017).

[50] *GraphLab*, 2013. [Online]. Available: https://turi.com/ (visited on 08/08/2017).

[51] Y. Guo, W. Bland, P. Balaji, and X. Zhou, "Fault tolerant MapReduce-MPI for HPC clusters", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, pp. 1–12.

[52] *Hadoop workload analysis*, Carnegie Mellon University, 2015. [Online]. Available: http://www.pdl.cmu.edu/HLA/index.shtml (visited on 09/01/2015).

[53] *The Apache Hadoop Project*, 2007. [Online]. Available: http://hadoop.apache.org/ (visited on 06/26/2017).

[54] *Powered by Hadoop*, 2017. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy/ (visited on 03/24/2017).

[55] *HDFS. The Hadoop Distributed File System*, 2007. [Online]. Available: http://hadoop.apache.org/common/docs/r1.2.1/hdfs_design.html (visited on 06/26/2017).

[56] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors", in *International Conference on Parallel Architectures and Compilation Techniques*, ACM, 2008, pp. 260–269.

[57] I. El-Helw, R. Hofman, and H. E. Bal, "Scaling MapReduce vertically and horizontally", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2014, pp. 525–535.

[58] *HiBench Big Data microbenchmark suite*, 2017. [Online]. Available: https://github.com/intel-hadoop/HiBench/ (visited on 07/01/2017).

[59] C.-H. Hsu, K. D. Slagter, and Y.-C. Chung, "Locality and loading aware virtual machine mapping techniques for optimizing communications in MapReduce applications", *Future Generation Computer Systems*, vol. 53, pp. 43–54, 2015.

[60] D. Huang, X. Shi, S. Ibrahim, L. Lu, H. Liu, S. Wu, and H. Jin, "Mr-scope: a real-time tracing tool for MapReduce", in *International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2010, pp. 849–855.

[61] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur-Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda, "High-performance design of HBase with RDMA over Infiniband", in *International Parallel and Distributed Processing Symposium*, IEEE, 2012, pp. 774–785.

[62] S. Ibrahim, "Performance-aware scheduling for data-intensive cloud computing", PhD thesis, HUST, 2011.

[63] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu, "Maestro: replica-aware map scheduling for MapReduce", in *International Symposium on Cluster, Cloud and Grid Computing*, IEEE/ACM, 2012, pp. 435–442.

[64] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "Leen: locality/fairness-aware key partitioning for MapReduce in the cloud", in *International Conference on Cloud Computing Technology and Science*, IEEE, 2010, pp. 17–24.

[65] S. Ibrahim, T.-D. Phan, A. Carpen-Amarie, H.-E. Chihoub, D. Moise, and G. Antoniu, "Governing energy consumption in Hadoop through CPU frequency scaling: an analysis", *Future Generation Computer Systems*, vol. 54, pp. 219–232, 2016.

[66] S. Ibrahim, T. A. Phuong, and G. Antoniu, "An eye on the elephant in the wild: a performance evaluation of Hadoop's schedulers under failures", in *International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, Springer, 2015, pp. 141–157.

[67] *IDC's Data Age 2025 study*, International Data Corporation, 2017. [Online]. Available: http://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf (visited on 06/25/2017).

[68] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole, "Enabling event tracing at leadership-class scale through I/O forwarding middleware", in *International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2012, pp. 49–60.

[69] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks", in *Special Interest Group on Operating Systems Review*, ACM, vol. 41, 2007, pp. 59–72.

[70] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters", in *Special Interest Group on Operating Systems Principles*, ACM, 2009, pp. 261–276.

[71] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance RDMA-based design of HDFS over InfiniBand", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012, p. 35.

[72] N. S. Islam, D. Shankar, X. Lu, M. Wasi-Ur-Rahman, and D. K. Panda, "Accelerating I/O performance of Big Data analytics on HPC clusters through RDMA-based key-value store", in *International Conference on Parallel Processing*, IEEE, 2015, pp. 280–289.

[73] N. S. Islam, M. Wasi-ur-Rahman, X. Lu, and D. K. Panda, "High performance design for HDFS with byte-addressability of NVM and RDMA", in *International Conference on Supercomputing*, ACM, 2016, pp. 1–14.

[74] W. Jiang, V. T. Ravi, and G. Agrawal, "A MapReduce system with an alternate API for multi-core environments", in *International Conference on Cluster, Cloud and Grid Computing*, IEEE, 2010, pp. 84–93.

[75] H. Jin, S. Ibrahim, T. Bell, W. Gao, D. Huang, and S. Wu, "Cloud types and services", in *Handbook of Cloud Computing*, Springer, 2010, pp. 335–355.

[76] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, "Quiet neighborhoods: key to protect job performance predictability", in *International Parallel and Distributed Processing Symposium*, IEEE, 2015, pp. 449–459.

[77] *Apache Kafka*, 2011. [Online]. Available: https://kafka.apache.org/ (visited on 08/08/2017).

[78] K. Kc and K. Anyanwu, "Scheduling Hadoop jobs to meet deadlines", in *International Conference on Cloud Computing Technology and Science*, IEEE, 2010, pp. 388–392.

[79] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun, "Leveraging burst buffer coordination to prevent I/O interference", in *International Conference on e-Science*, IEEE, 2016, pp. 371–380.

[80] *Kraken Cray XT5 system*, National Institute for Computational Sciences, 2009. [Online]. Available: http://www.nics.tennessee.edu/computing-resources/kraken/ (visited on 08/04/2017).

[81] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: stream processing at scale", in *International Conference on Management of Data*, ACM, 2015, pp. 239–250.

[82] C.-S. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf, "How file access patterns influence interference among cluster applications", in *International Conference on Cluster Computing*, IEEE, 2014, pp. 185–193.

[83] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa, "I/O scheduling service for multi-application clusters", in *International Conference on Cluster Computing*, IEEE, 2006, pp. 1–10.

[84] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: reliable, memory speed storage for cluster computing frameworks", in *International Symposium on Cloud Computing*, ACM, 2014, pp. 1–15.

[85] Z. Li and H. Shen, "Designing a hybrid scale-up/out Hadoop architecture based on performance measurements for high application performance", in *International Conference on Parallel Processing*, IEEE, 2015, pp. 21–30.

[86] T. Lippert, D. Mallmann, and M. Riedel, "Scientific Big Data analytics by HPC", in *John von Neumann Institute for Computing Symposium*, Jülich Supercomputing Center, 2016.

[87] H. Liu and B. He, "Reciprocal resource fairness: towards cooperative multiple-resource fair sharing in IAAS clouds", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2014, pp. 970–981.

[88] N. 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 *International Symposium on Mass Storage Systems and Technologies*, IEEE, 2012, pp. 1–11.

[89] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the I/O performance of petascale storage systems", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2010, pp. 1–12.

[90] I. Lopez, *IDC talks convergence in high performance data analysis*, 2013. [Online]. Available: https://www.datanami.com/2013/06/19/idc_talks_convergence_in_high_performance_data_analysis/ (visited on 06/15/2017).

[91] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-performance design of Hadoop RPC with RDMA over InfiniBand", in *International Conference on Parallel Processing*, IEEE, 2013, pp. 641–650.

[92] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "Datampi: extending MPI to Hadoop-like Big Data computing", in *International Parallel and Distributed Processing Symposium*, IEEE, 2014, pp. 829–838.

[93] D. Luan, S. Huang, and G. Gong, "Using Lustre with Apache Hadoop", Sun Microsystems Inc, Tech. Rep., 2009.

[94] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing MapReduce for multicore architectures", Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep., 2010.

[95] A. Marathe, R. Harris, D. Lowenthal, B. R. De Supinski, B. Rountree, and M. Schulz, "Exploiting redundancy for cost-effective, time-constrained execution of HPC applications on Amazon EC2", in *International Symposium on High-performance Parallel and Distributed computing*, ACM, 2014, pp. 279–290.

[96] P. Mell, T. Grance, *et al.*, "The NIST definition of cloud computing, Recommendations of the national institute of standards and technology", National Institute of Standards and Technology, Tech. Rep., 2011.

[97] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2010, pp. 1–11.

[98] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu, "Going back and forth: efficient multideployment and multisnapshotting on clouds", in *International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2011, pp. 147–158.

[99] Z. Niu, S. Tang, and B. He, "Gemini: an adaptive performance-fairness scheduler for data-intensive cluster computing", in *International Conference on Cloud Computing Technology and Science*, IEEE, 2015, pp. 66–73.

[100] *OpenCloud Hadoop cluster trace: format and schema*, Carnegie Mellon University, 2012. [Online]. Available: http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html (visited on 02/01/2017).

[101] M. Pastorelli, M. Dell'Amico, and P. Michiardi, "OS-assisted task preemption for Hadoop", in *International Conference on Distributed Computing Systems Workshops*, IEEE, 2014, pp. 94–99.

[102] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems", in *Conference on File and Storage Technologies*, USENIX, vol. 8, 2008, pp. 1–14.

[103] J. Polo, D. Carrera, Y. Becerra, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for MapReduce environments", in *Network Operations and Management Symposium*, IEEE, 2010, pp. 373–380.

[104] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: sharing the network in cloud computing", in *International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ACM, 2012, pp. 187–198.

[105] R. Power and J. Li, "Piccolo: building fast, distributed programs with partitioned tables", in *International Symposium on Operating Systems Design and Implementation*, USENIX, vol. 10, 2010, pp. 1–14.

[106] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger, "A novel network request scheduler for a large scale storage system", *Computer Science-Research and Development*, vol. 23, no. 3, pp. 143–148, 2009.

[107] J.-A. Quiané-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "RAFTing MapReduce: fast recovery on the RAFT", in *International Conference on Data Engineering*, 2011, pp. 589–600.

[108] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads", *International Journal on Very Large Databases*, vol. 6, no. 10, pp. 853–864, 2013.

[109] A. Rosà, L. Y. Chen, R. Birke, and W. Binder, "Demystifying casualties of evictions in Big Data priority scheduling", *SIGMETRICS Performance Evaluation Review*, vol. 42, no. 4, pp. 12–21, 2015.

[110] R. B. Ross, R. Thakur, *et al.*, "PVFS: a parallel file system for Linux clusters", in *Annual Linux Showcase and Conference*, 2000, pp. 391–430.

[111] K. Salem and H. Garcia-Molina, "Checkpointing memory-resident databases", in *International Conference on Data Engineering*, IEEE, 1989, pp. 452–462.

[112] *Apache Samza*, 2014. [Online]. Available: https://samza.apache.org/ (visited on 08/08/2017).

[113] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in Hadoop", in *Job Scheduling Strategies for Parallel Processing*, Springer, 2010, pp. 110–131.

[114] 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 *International Symposium on Cluster, Cloud and Grid Computing*, IEEE, 2014, pp. 21–30.

[115] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance", *International Journal on Very Large Databases*, vol. 3, no. 1-2, pp. 460–471, 2010.

[116] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers", *Journal of Physics: Conference Series*, vol. 78, no. 1, pp. 12–22, 2007.

[117] P. Schwan *et al.*, "Lustre: building a file system for 1000-node clusters", in *Annual Linux Symposium*, vol. 2003, 2003, pp. 380–386.

[118] S. Sehrish, G. Mackey, J. Wang, and J. Bent, "MRAP: a novel MapReduce-based framework to support HPC analytics applications with access patterns", in *International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2010, pp. 107–118.

[119] H. Shan and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms", Lawrence Berkeley National Laboratory, Tech. Rep., 2007.

[120] E. Smirni and D. A. Reed, "Lessons from characterizing the input/output behavior of parallel scientific applications", *Performance Evaluation*, vol. 33, no. 1, pp. 27–44, 1998.

[121] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-side I/O coordination for parallel file systems", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2011, pp. 1–11.

[122] *Apache Spark primer*, Databricks, 2017. [Online]. Available: http://go.databricks.com/hubfs/pdfs/Apache_Spark_Primer_170303.pdf (visited on 07/30/2017).

[123] *Apache Storm*, 2012. [Online]. Available: https://storm.apache.org/ (visited on 06/27/2017).

[124] S. Tang, B.-s. Lee, B. He, and H. Liu, "Long-term resource fairness: towards economic fairness on pay-as-you-use computing systems", in *International Conference on Supercomputing*, ACM, 2014, pp. 251–260.

[125] Y. Tanimura, R. Filgueira, I. Kojima, and M. Atkinson, "Reservation-based I/O performance guarantee for MPI-IO applications using shared storage systems", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012, pp. 1382–1383.

[126] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the duality of data-intensive file system design: reconciling HDFS and PVFS", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2011, pp. 1–12.

[127] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody, "Managing I/O interference in a shared burst buffer system", in *International Conference on Parallel Processing*, IEEE, 2016, pp. 416–425.

[128] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a MapReduce framework", *International Journal on Very Large Databases*, vol. 2, no. 2, pp. 1626–1629, 2009.

[129] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, and M. Valero, "Spark deployment and performance evaluation on the MareNostrum supercomputer", in *International Conference on Big Data*, IEEE, 2015, pp. 299–306.

[130] *Trinity project*, Los Alamos National Laboratory, 2015. [Online]. Available: http://www.lanl.gov/projects/trinity/index.php/ (visited on 08/04/2017).

[131] R. Tudoran, A. Costan, G. Antoniu, and H. Soncu, "Tomusblobs: towards communication-efficient storage for MapReduce applications in Azure", in *International Symposium on Cluster, Cloud and Grid Computing*, IEEE, 2012, pp. 427–434.

[132] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache Hadoop YARN: yet another resource negotiator", in *Annual Symposium on Cloud Computing*, ACM, 2013, pp. 1–16.

[133] C. Vecchiola, R. N. Calheiros, D. Karunamoorthy, and R. Buyya, "Deadline-driven provisioning of resources for scientific applications in hybrid clouds with Aneka", *Future Generation Computer Systems*, vol. 28, no. 1, pp. 58–65, 2012.

[134] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling", in *International Conference on Operating Systems Design and Implementation*, USENIX Association, 2014, pp. 301–316.

[135] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability", in *Symposium on Cloud computing*, ACM, 2010, pp. 193–204.

[136] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii, "Accelerating I/O forwarding in IBM Blue Gene/P systems", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2010, pp. 1–10.

[137] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "TRIO: burst buffer based I/O orchestration", in *International Conference on Cluster Computing*, IEEE, 2015, pp. 194–203.

[138] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "Burstmem: a high-performance burst buffer system for scientific applications", in *International Conference on Big Data*, IEEE, 2014, pp. 71–79.

[139] Y. Wang, R. Goldstone, W. Yu, and T. Wang, "Characterization and optimization of memory-resident MapReduce on HPC systems", in *International Parallel and Distributed Processing Symposium*, IEEE, 2014, pp. 799–808.

[140] Y. Wang, J. Tan, W. Yu, X. Meng, and L. Zhang, "Preemptive reducetask scheduling for fair and fast job completion", in *International Conference on Autonomic Computing*, 2013, pp. 279–289.

[141] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang, "Smart: a MapReduce-like framework for in-situ scientific analytics", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, pp. 1–12.

[142] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: a resilient distributed graph system on Spark", in *International Workshop on Graph Data Management Experiences and Systems*, ACM, 2013, pp. 1–6.

[143] P. Xuan, J. Denton, P. K. Srimani, R. Ge, and F. Luo, "Big Data analytics on traditional HPC infrastructure using two-level storage", in *International Workshop on Data-Intensive Scalable Computing Systems*, ACM, 2015, pp. 1–8.

[144] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the root causes of cross-application I/O interference in HPC storage systems", in *International Parallel and Distributed Processing Symposium*, IEEE, 2016, pp. 750–759.

[145] O. Yildiz, S. Ibrahim, and G. Antoniu, "Enabling fast failure recovery in shared Hadoop clusters: towards failure-aware scheduling", *Future Generation Computer Systems*, vol. 74, pp. 208–219, 2016.

[146] O. Yildiz, S. Ibrahim, T. A. Phuong, and G. Antoniu, "Chronos: failure-aware scheduling in shared Hadoop clusters", in *International Conference on Big Data*, IEEE, 2015, pp. 313–318.

[147] O. Yildiz, A. C. Zhou, and S. Ibrahim, "Eley: on the effectiveness of burst buffers for Big Data processing in HPC systems", in *International Conference on Cluster Computing*, IEEE, 2017, pp. 87–91.

[148] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: scalable MapReduce on a large-scale shared-memory system", in *International Symposium on Workload Characterization*, IEEE, 2009, pp. 198–207.

[149] Z. Yu, M. Li, X. Yang, H. Zhao, and X. Li, "Taming non-local stragglers using efficient prefetching in MapReduce", in *International Conference on Cluster Computing*, IEEE, 2015, pp. 52–61.

[150] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling", in *European Conference on Computer systems*, ACM, 2010, pp. 265–278.

[151] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing", in *International Conference on Networked Systems Design and Implementation*, USENIX, 2012, pp. 15–28.

[152] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets", in *International Workshop on Hot Topics in Cloud Computing*, USENIX, vol. 10, 2010, pp. 1–7.

[153] F. Zanon Boito, R. Kassick, P. O. A. Navaux, and Y. Denneulin, "AGIOS: application-guided I/O scheduling for parallel file systems", in *International Conference on Parallel and Distributed Systems*, IEEE, 2013, pp. 43–50.

[154] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: improving the performance of multi-node I/O systems via inter-server coordination", in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE/ACM, 2010, pp. 1–11.

[155] X. Zhang and S. Jiang, "InterferenceRemoval: removing interference of disk access for MPI programs through data replication", in *International Conference on Supercomputing*, ACM, 2010, pp. 223–232.

[156] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "Predata–preparatory data analytics on peta-scale machines", in *International Parallel and Distributed Processing Symposium*, IEEE, 2010, pp. 1–12.

[157] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/O-aware batch scheduling for petascale computing systems", in *International Conference on Cluster Computing*, IEEE, 2015, pp. 254–263.

Chapter 8

Appendix

Contexte

En 2017 nous vivons dans un monde régi par les données. Les applications d'analyse de données apportent des améliorations fondamentales dans de nombreux domaines tels que les sciences, la santé et la sécurité. Cela a stimulé la croissance des volumes de données (le déluge du Big Data). Par exemple, l'International Data Corporation (IDC) Research rapporte que la quantité de données générées en 2016 était de 16,1 zettabytes. La même étude prévoit que 163 zettabytes seront engendrées en 2025, ce qui se traduit par une augmentation de dix fois dans la taille des données en moins d'une décennie.

Pour extraire des informations utiles à partir de cette quantité énorme d'informations, différents modèles de traitement des données ont émergé. Parmi eux, MapReduce s'est imposé comme le modele dominant, en particulier pour le traitement par lots (batch processing). MapReduce, et sa mise en œuvre open source Hadoop, ont été adoptés à la fois dans l'industrie et dans le milieu universitaire en raison de leur simplicité, de leur tolérance aux pannes transparente et de leur évolutivité. Par exemple, Carnegie Mellon University (CMU) utilise des clusters Hadoop pour l'analyse de données dans plusieurs domaines scientifiques, y compris l'astrophysique, la neurolinguistique, le traitement du langage naturel, la bioinformatique et l'analyse des réseaux sociaux. Outre MapReduce et Hadoop, de nouveaux outils de traitement de données ont été introduits, tels que Spark et Storm. Ces derniers se focalisent sur les applications itératives, le traitement des flux, l'apprentissage automatique et le traitement de graphes. Nous appelons ces applications des *"applications Big Data"*.

Les traitements Big Data est traditionnellement exécuté sur les Clouds. Les Clouds fournissent des ressources à grande échelle à moindre coût. Par conséquent, de nombreuses industries et laboratoires de recherche utilisent des Clouds à grande échelle pour faire face aux volumes de données gigantesques. Par exemple, Google emploie un cluster avec plus de 200

000 cœurs pour délivrer ses services aux entreprises et ainsi améliorer la qualité d'expérience de l'utilisateur.

Outre les Clouds, les systèmes HPC (High-Performance Computing) ont suscité un grand intérêt de par leur capacité à effectuer efficacement des traitements Big Data. Les systèmes HPC sont équipés de réseaux haut-débit et de milliers de nœuds avec de nombreux cœurs, ce qui leur profère un grand potentiel pour les traitements Big Data, en particulier pour les applications nécessitant un temps de réponse court. Par exemple, PayPal a récemment déployé son logiciel de détection de fraude sur des systèmes HPC afin de détecter rapidement les fraudes parmi des millions de transactions. Par ailleurs, les applications Big Data peuvent faciliter la réalisation de simulations à grande échelle sur les systèmes HPC (en filtrant / analysant les données de simulation). Par exemple, l'institut John von Neumann Institute for Computing essaie de coupler des applications Big Data avec des simulations scientifiques afin de relever les grands défis sociétaux et scientifiques.

Habituellement, ces plateformes à grande échelle (les systèmes HPC et les Clouds) sont utilisées simultanément par plusieurs utilisateurs et de multiples applications afin d'optimiser l'utilisation des ressources. Par exemple, une soixantaine de jobs sont executés simultanément sur le supercalculateur Intrepid d'Argonne. Bien qu'il y ait beaucoup d'avantages à partager de ces plates-formes, plusieurs problèmes sont soulevés dès lors qu'un nombre important d'utilisateurs et d'applications les utilisent en même temps. Par exemple, la gestion des Entrées/Sorties (E/S) et des pannes est au cœur d'un traitement efficace des données sur ces plateformes.

Dans le contexte des systèmes HPC, un grand nombre d'applications Big Data fonctionnera simultanément sur la même plateforme et partagera le système de stockage (système de fichiers parallèle). Ainsi, les interférences d'E/S apparaîtront comme un problème majeur pour la performance de ces applications. L'interférence E/S est un problème bien connu dans les systèmes HPC, il empêche souvent les applications d'utiliser pleinement les ressources mises à leur disposition. D'autre part, les applications Big Data seront confrontées à des latences élevées lors de la réalisation d'E/S en raison des transferts de données nécessaires entre le système de fichiers parallèle et les noeuds de calcul. D'ailleurs, l'impact des interférences et de la latence sera amplifié lorsque les systèmes HPC serviront de plateforme sous-jacente pour les applications Big Data, ainsi que pour les simulations HPC à grande échelle. De plus, l'exécution d'applications Big Data sur ces systèmes peut gravement dégrader les performances de ces simulations qui sont considérées comme des citoyens de première classe sur les systèmes HPC. Cela soulève le défi de pouvoir effectuer un traitement efficace de données sans affecter les performances des simulations à grande échelle qui seraient en cours d'exécution.

Enfin, les pannes sont inévitables dans les plateformes à grande échelle, en particulier dans les Clouds. C'est parce que les Clouds se composent de machines fortement susceptibles de tomber en panne, ce qui rend les défaillances une réalité quotidienne. Par exemple, Barroso et al. ont indiqué que le temps moyen entre deux pannes dans un cluster de 10 000 machines est de quelques heures. Par conséquent, le recouvrement rapide des pannes joue un rôle crucial pour le traitement efficace de données dans les Clouds.

Contributions

Cette thèse se concentre sur les facteur qui déterminent la performance des traitements Big Data sur les plateformes partagées à grande échelle. Notre objectif est de fournir des solutions de gestion des E/S et de recouvrement de pannes qui peuvent améliorer le la performance des traitements des données sur ces plateformes.

Nous nous concentrons tout d'abord sur les goulots d'étranglement liés aux performances des E/S pour les applications Big Data sur les systèmes HPC. Nous commençons par caractériser les performances des applications Big Data sur ces systèmes. Nous identifions les interférences et la latence des E/S comme les principaux facteurs limitant les performances. Ensuite, nous nous intéressons de manière plus détaillée aux interférences des E/S afin de mieux comprendre les causes principales de ce phénomène. De plus, nous proposons un système de gestion des E/S pour réduire les latences que les applications Big Data peuvent subir sur les systèmes HPC. Aussi, nous introduisons des modèles d'interférence pour les applications Big Data et HPC en fonction des résultats que nous obtenons dans notre étude expérimentale concernant les causes des interférences d'E/S. Enfin, nous exploitons ces modèles afin de minimiser l'impact des interférences sur les performances des applications Big Data et HPC.

Deuxièmement, nous nous concentrons sur l'impact des défaillances sur la performance des applications Big Data en étudiant la gestion des pannes dans les clusters MapReduce partagés. Nous présentons un ordonnanceur qui permet un recouvrement rapide des pannes, améliorant ainsi les performances des applications Big Data.

Ces contributions peuvent être résumées comme suit.

Caractérisation de la performance des applications Big Data sur les systèmes HPC

Il existe un intérêt récent pour l'exécution de traitements Big Data sur les systèmes HPC. Bien que ces systèmes (HPC) offrent un vaste éventail d'opportunités pour le traitement de données massives (réseaux haut-débit, multi-cœurs, stockage à accès très rapide), ils sont traditionnellement conçus pour des applications de calcul intensif et non pour gérer des données massives. Par conséquent, il est nécessaire de comprendre les caractéristiques et les performances des applications Big Data sur les systèmes afin de pouvoir les utiliser efficacement. Pour ce faire, nous avons mené une étude expérimentale pour mieux comprendre les performances de Spark, le logiciel de traitement de données en mémoire de facto, sur les systèmes HPC. Nous avons exécuté Spark en utilisant des charges de travail représentatives du Big Data sur Grid'5000 pour évaluer comment la latence, la contention et la configuration du système de fichiers peuvent influer sur les performances des applications. Nos résultats expérimentaux révèlent que les interférences et que les latences d'E/S sont les facteurs majeurs limitant la performance lors de l'exécution d'applications Big Data sur les systèmes HPC. Nous discutons également des implications de ces résultats et attirons l'attention sur les solutions de gestion des E/S (Burst Buffers) pour améliorer les performances des applications Big Data sur les systèmes HPC.

Identifier les causes premières des interférences des E/S sur les systèmes HPC

Comme les systèmes HPC sont partagés par un grand nombre d'applications en même temps, les interférences d'E/S apparaissent comme un facteur limitant pour les performances des applications Big Data. Les interférences des E/S, qui est un problème bien connu dans l'HPC, sont définies comme la dégradation des performances observées par une application unique exécutant une E/S en conflit avec d'autres applications s'exécutant sur la même plateforme. Ce problème d'interférence devient de plus en plus important avec le nombre croissant d'applications (des applications HPC et Big Data) qui partagent ces plateformes. Notre but est d'effectuer un traitement intelligent prenant en compte les interférences pour des applications Big Data sur ces plateformes. Dans cette optique, la compréhension des causes premières du problème d'interférence des E/S devient cruciale. À cet effet, nous avons mené une vaste étude expérimentale en utilisant de micro-applications sur Grid'5000 pour évaluer comment le modèle d'accès des applications, les composants réseau, la configuration du système de fichiers et les périphériques de stockage backend influencent les interférences des E/S. Notre étude révèle que, dans de nombreuses situations, l'interférence est le résultat d'un mauvais contrôle de flux dans le chemin des E/S et n'est pas aussi souvent due à la limitation d'un composant dans le chemin des E/S. Nous montrons aussi que le comportement sans interférences n'est pas nécessairement un signe de performance optimale. À notre connaissance, notre travail fournit la première description détaillée du rôle de chacune des causes premières potentielles des interférences et de leur interactions. Ce travail, partiellement effectué durant un stage de 3 mois à ANL, a mené à une publication dans la conférence IPDPS 2016.

Proposition d'une architecture à base de burst buffers pour des systèmes HPC, permettant d'accélérer les traitements de données

Le développement de solutions intelligentes de gestion des E/S est d'une importance capitale pour effectuer un traitement efficace des données sur les systèmes HPC. Les tampon de pic (Burst Buffers : BB) sont une solution efficace pour réduire le temps de transfert de données et des interférences des E/S sur les systèmes HPC. L'extension des BB pour gérer les applications Big Data est difficile car ils doivent tenir compte des grandes tailles de données et de la qualité de service (QoS, contraintes de performance définies par l'utilisateur) des applications HPC - considérées comme des citoyens de première classe sur les systèmes HPC. Les BB existants se concentrent uniquement sur les données intermédiaires et entraînent une dégradation élevée des applications Big Data et HPC. Nous proposons *Eley*, une solution basée sur des BB qui aide à accélérer les performances des applications Big Data tout en garantissant la QoS des applications HPC. Pour atteindre cet objectif, Eley intègre une technique de prélecture sensible aux interférences qui rend la lecture des données d'entrée plus rapide tout en contrôlant les interférences imposées par les applications Big Data sur les applications HPC. Nous avons évalué expérimentalement Eley avec un système réel sur Grid'5000 et des simulations utilisant une large gamme d'applications Big Data et HPC. Nos résultats démontrent l'efficacité d'Eley dans la réduction du temps d'exécution des applications Big Data tout en maintenant la QoS des applications HPC. Une partie de ce travail a été publiée lors de la conférence CLUSTER'17.

Recouvrement rapide des défaillances sur des systèmes partagés à grande échelle

Les défaillances sont assez courantes dans les plateformes à grande échelle, en particulier pour les Clouds qui se composent de matériel de grande série. Ainsi, la gestion des défaillances joue un rôle crucial pour un traitement Big Data efficace. Dans ce sens, nous visons à améliorer la performance des applications en cas des défaillances sur les clusters Hadoop partagés. Actuellement, Hadoop gère les défaillances de la machine en exécutant de nouveau toutes les tâches des machines défectueuses (en exécutant des tâches de recouvrement). Malheureusement, cette solution élégante est entièrement confinée au cœur d'Hadoop et est cachée aux ordonnanceurs. L'inconscience des défaillances peut donc empêcher les ordonnanceurs Hadoop de fonctionner correctement pour atteindre leurs objectifs (l'équité, la priorité du travail) et peuvent avoir une incidence importante sur les performances des applications Big Data. Nous avons abordé ce problème en proposant *Chronos*, une stratégie d'ordonnancement qui permet une action rapide et intelligente pour un recouvrement rapide après défaillance, tout en respectant un objectif de l'ordonnanceur. Lors de la détection d'une défaillance, plutôt que d'attendre un certain temps pour obtenir des ressources pour les tâches de recouvrement, Chronos s'appuie sur une technique légère de préemption pour répartir ces ressources avec précaution. En outre, Chronos considère la localisation des données lors d'ordonnancement des tâches de recouvrement pour améliorer les performances. Nous démontrons l'utilité de Chronos en la combinant avec les ordonnanceurs FIFO et Fair. Les résultats expérimentaux montrent que Chronos reprend un comportement d'ordonnancement correct en quelques secondes seulement et réduit les temps d'exécution du travail jusqu'à 55% par rapport aux ordonnanceurs présents dans la littérature. Ce travail a mené à des publications à la conférence BigData'15 et au journal FGCS.

List of Publications

Journals

- **Orcun Yildiz**, Shadi Ibrahim, Gabriel Antoniu. *Enabling Fast Failure Recovery in Shared Hadoop Clusters : Towards Failure-Aware Scheduling*, Journal of the Future Generation Computer Systems **(FGCS)**, 2016.
- Matthieu Dorier, **Orcun Yildiz**, Shadi Ibrahim, Anne-Cécile Orgerie, Gabriel Antoniu. *On the Energy Footprint of I/O Management in Exascale HPC Systems*, Journal of the Future Generation Computer Systems **(FGCS)**, 2016.
- Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, **Orcun Yildiz**, Shadi Ibrahim, Tom Peterka, Leigh Orf. *Damaris : Addressing Performance Variability in Data Management for Post-Petascale Simulations*, ACM Transactions on Parallel Computing **(TOPC)**, 2016.

International Conferences

- **Orcun Yildiz**, Amelie Chi Zhou, Shadi Ibrahim. *Eley : On the Effectiveness of Burst Buffers for Big Data Processing in HPC systems*, in Proceedings of the 2017 IEEE International Conference on Cluster Computing **(CLUSTER '17)**, *Hawaii, September 2017.* CORE Rank A (acceptance rate 31%).
- **Orcun Yildiz**, Matthieu Dorier, Shadi Ibrahim, Rob Ross, Gabriel Antoniu. *On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems*, in Proceedings of the 2016 IEEE International Parallel & Distributed Processing Symposium **(IPDPS '16)**, *Chicago, May 2016.* CORE Rank A (acceptance rate 23%).
- **Orcun Yildiz**, Shadi Ibrahim, Tran Anh Phuong, Gabriel Antoniu. *Chronos : Failure-Aware Scheduling in Shared Hadoop Clusters*, in Proceedings of the 2015 IEEE International Conference on Big Data **(BigData '15)**, *Santa Clara, October 2015.* (acceptance rate 35%).

Workshops at International Conferences

- **Orcun Yildiz**, Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu. *A Performance and Energy Analysis of I/O Management Approaches for Exascale Systems*, in Proceedings of the 2014 Data-Intensive Distributed Computing **(DIDC '14)** workshop, held in conjunction with the 23rd International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC '14), *Vancouver, June 2014.*