University of California
Santa Barbara

Resource Allocation in Multi-analytics, Resource-Constrained Environments

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Stratos Dimopoulos

Committee in charge:

Professor Chandra Krintz, Co-Chair
Professor Rich Wolski, Co-Chair
Professor Tevfik Bultan

January 2018

The Dissertation of Efstratios Dimopoulos is approved.

Professor Rich Wolski

Professor Tevfik Bultan

Professor Chandra Krintz, Committee Chair

January 2018

Resource Allocation in Multi-analytics, Resource-Constrained Environments

Efstratios Dimopoulos

Dedicated to my parents and my brother for their unconditional love.

Acknowledgements

My sincere gratitude goes out to my advisors Chandra Krintz and Rich Wolski for their continuous support and advice during my PhD journey. They are two of the most enthusiastic and generous people I know, with a very broad spectrum of knowledge, and inspiring professional careers and lifestyles. I consider myself privileged for participating in countless intense conversations with them that helped me grow both as a researcher and as a person. Among so many other things, they worked hard with me to teach me how to identify research questions and come up with feasible plans to address them, how to effectively present research on a paper, how to communicate ideas, and how to methodologically work and make progress despite any setbacks and disappointments.

I also thank Tevfik Bultan for his great research feedback and for honoring me by being a member on my PhD committee. I am grateful to all UCSB faculty who gave their advice during these years and specifically to Amr El Abbadi, Elizabeth Belding, Rachel Lin, Tim Sherwood, Stefano Tessaro, and Xifeng Yan. I am also grateful to my teachers and mentors from the past, Alex Delis, Ioannis Ioannidis and Lazaros Merakos, who generously advised me and helped me enter the PhD program. My special thanks go out to George Polyzos for inspiring me and encouraging me to follow the PhD path and for constantly supporting me ever after.

A major thanks goes to my sweet family for their unconditional love and constant support throughout my life. My parents prioritize my own dreams above theirs and are always willing to go through any sacrifice to help me achieve them. I am also grateful to my amazing brother who encouraged me to take this big step, care, and advised me throughout this process. He and his family are an abundant source of love. I owe to everyone in my family, for the values they distilled in me and for never letting me feel alone. I thank them for tolerating my absence and sharing my worries. Their strong

faith in my abilities is a constant motivation to never stop trying and always giving my best.

I am thankful to the amazing new friends I made in Santa Barbara but also to the ones I left behind for being an abundant emotional support. I thank Meni, Theodore, Kyriakos, and Stathis for becoming my best friends in Santa Barbara. I will never forget our countless fun times and talks we had. I am grateful to my beloved old friends Georgia, Lefteris, Manolis, Elena, and Petros for making me feel a special part of their lives despite the long distance that physically separates us. I am privileged to have in close proximity my childhood friends Kostas and Ioanna and I thank them for their hospitality, their advice, and encouragement. I am lucky to be surrounded these past 6 years with an amazing group of people from all over the word and I thank them for the fun times we had. Time in Santa Barbara without Adam, Anahita, Emma, Haraldur, Jacob, Ivan, Leah, Lena, Lina, Karim, Melika, Sergio, Xiaofei, and the gang from Sec Lab wouldn't feel the same. Lastly, a special thanks goes out to the people of Atnet and all my good friends, Alexandra, Alexis, Andreas, Babis, Efi, Giannis, Giorgos, Lorena, Manos, Maria, Nontas, Panagiotis, Paula, Sofia, Stef, and Tavouli who overlooked the distance and found ways to remain close to me and support me until the very end of this journey.

I am grateful to have lab mates and colleagues that did not only provide insightful research feedback but also pleasant talks, and great friendship. I thank Alex for being my occasional gym-partner, an always supportive lab mate and a good friend, Nevena for our friendship, chocolate-sharing, and our countless talks and laughs, Hiranya, an inspirational researcher who is always a great source of advice, Alexandros, Angad, Benji, Cetin, Chris, Kevin, Kyle, Lara, Paz, Vaibhav, Victor, and Wei-Tsung for their good advice and friendship they provided when I needed it the most.

Lastly, I want to thank all the people that collaborated with me or devoted time

to discuss my research problems. I am very grateful to the people from Altiscale and Turn, and especially Abin Shahab, Cosmin Negruseri, Eric Wohlstadter, Hazem Elmeleegy, Raymie Stata, Ricardo Jenez, Thomas Nystrand, and Zoran Dimitrijevic for their mentorship and support on my research. I thank Vivek Thakre, Wahid Chrabakh, and Luca Foschini who generously provided their help on my first steps with big data analytics systems and data science. Finally, I thank the researchers and patients at William Sansum Diabetes Center and our collaborators at the Chemical Engineering department at UCSB for giving me the chance to work on an inspiring research project.

Curriculum Vitæ

Efstratios Dimopoulos

Education

2017	Ph.D. in Computer Science (Expected), University of California, Santa Barbara, United States.
2007	M.S. in Computer Science, Athens University of Economics and Business, Greece
2005	B.S in Informatics and Telecommunications University of Athens, Greece

Publications

[Best Paper Award]
Justice: A Deadline-aware, Fair-share Resource Allocator for Implementing Multi-analytics
S. Dimopoulos, C. Krintz, and R. Wolski
IEEE Cluster Computing Conference, September 2017

PYTHIA: Admission Control for Multi-Framework, Deadline-Driven, Big Data Workloads
S. Dimopoulos, C. Krintz, and R. Wolski
IEEE International Conference on Cloud Computing, June 2017

Big Data Framework Interference In Restricted Private Cloud Settings
S. Dimopoulos, C. Krintz, and R. Wolski
IEEE International Conference on Big Data, December, 2016

On the Use of Consumer-grade Activity Monitoring Devices to Improve Predictions of Glycemic Variability
C. Krintz, R. Wolski, J. E. Pinsker, S. Dimopoulos, J. Brevik, and E. Dassau
EAI International Conference on Smart Wearable Devices and IoT for Health and Wellbeing Applications, October 2015.

SuperContra: Cross-Language, Cross-Runtime Contracts As a Service,
Stratos Dimopoulos, Chandra Krintz, Rich Wolski, and Anand Gupta
IC2E Workshop on the Future of PaaS, March 2015

Cloud Platform Support for API Governance
Chandra Krintz, Hiranya Jayathilaka, Stratos Dimopoulos, Alexander Pucher, Rich Wolski and Tevfik Bultan

IC2E Workshop on the Future of PaaS, March 2014

Developing Systems for API Governance
Chandra Krintz, Hiranya Jayathilaka, Stratos Dimopoulos, Alexander Pucher, and Rich Wolski
Workshop on Sustainable Software for Science: Practice and Experiences, 2013

Time-Shifting Traffic to Improve Utilization in Rural Area Networks
Ana Nika, Stratos Dimopoulos, David L. Johnson, Elizabeth M. Belding
CWIC SoCal, Santa Ana, USA, April 2012

Exploiting super peers for large-scale peer-to-peer Wi-Fi roaming
E.G. Dimopoulos, P.A. Frangoudis and G.C. Polyzos
Proc. IEEE Globecom 2010 Workshop on Advances in Communications and Networks, Miami, Florida, USA, December 2010

Abstract

Resource Allocation in Multi-analytics, Resource-Constrained Environments

by

Efstratios Dimopoulos

The vast proliferation of monitoring and sensing devices equipped with Internet connectivity, commonly known as the "Internet of Things" (IoT) generates an unprecedented volume of data, which requires Big Data Analytics Systems (BDAS) to process it and extract actionable insights. The large diversity of IoT data processing applications require the deployment of multiple processing frameworks under the coordination of a resource allocator. To enable prompt actuation, these applications must meet deadlines and their processing takes place near where data is generated, in private clouds or edge computing clusters, which have limited resources.

In resource-constrained and multi-analytics settings there are issues related to the combined use of open-source BDAS, originally designed for resource-rich, standalone clusters, that remain unaddressed. Specifically, open-source BDAS have unknown behavior when used combined under the coordination of a cluster-manager and the available resources are limited. Moreover, existing allocation policies are not suitable to meet deadlines in resource-constrained settings without wasting resources or requiring particular repetitive job patterns. Lastly, in such settings fair-share policies cannot reliably preserve fairness.

To satisfy deadlines and achieve allocation fairness in resource constrained clusters for multi-analytics, we employ predictive resource allocation and admission control. We evaluate the performance and behavior of BDAS in resource-constrained multi-analytics clusters and understand the root causes of their interference. Moreover, we design ad-

mission control and resource allocation suitable for resource-managers. Allocation decisions adapt to changing cluster conditions to satisfy deadlines and preserve fairness under resource-constrained multi-analytics settings. We evaluate our approach with trace-based simulations and production workloads and show that it satisfies more deadlines, preserves fairness, and utilizes the cluster more efficiently compared to existing fair-share allocators designed for resource managers.

Contents

Curriculum Vitae **viii**

Abstract **x**

1 Introduction **1**

2 Background **8**
2.1 Distributed Big Data Analytics Systems 8
2.2 Multi-Analytics 10
2.3 Resource Managers 11
2.4 Real time systems 12

3 Performance Interference of Multi-tenant, Big Data Frameworks in Resource Constrained Private Clouds **15**
3.1 Resource Sharing on Mesos 16
3.2 Experimental Methodology 21
3.3 Results 24
3.4 Related Work 37
3.5 Key Insights 39
3.6 Summary 41

4 PYTHIA: Admission Control for Multi-Framework, Deadline-Driven, Big Data Workloads **42**
4.1 PYTHIA Design 44
4.2 Experimental Methodology 48
4.3 Workload Characterization 50
4.4 Empirical Evaluation 52
4.5 Related Work 57
4.6 Summary 60

5 Justice: A Deadline-aware, Fair-share Resource Allocator for Implementing Multi-analytics **62**
5.1 Justice Design 64
5.2 Experimental Methodology 70
5.3 Workload Characterization 73
5.4 Results 75
5.5 Related Work 88
5.6 Summary 90

6 Conclusion **93**

Bibliography **99**

Chapter 1

Introduction

Recently, there has been explosive growth in the generation and production of data products and in the number and type of applications that process data. This is the result of the vast proliferation of environmental and personal monitoring and sensing devices equipped with Internet connectivity, commonly known as the "Internet of Things" (IoT), coupled with the widely adopted cloud-computing, which offers low cost compute and storage. Software engineers, data scientists and analysts develop and deploy all form of data mining, machine learning, stream processing, statistical analysis, image processing and other applications to extract actionable insights from this data and enable decision support. Such applications, can impact all aspects of our society and economy including social networking, e-commerce, health care, education, agriculture, communications, transportation and others.

To address the increased data processing needs, high-scalable, fault-tolerant, batch and stream processing frameworks have emerged. Such frameworks are widely accessible to the masses through open-source implementations (e.g., Apache Hadoop [1], Spark [2], Storm [3]). Stream processing frameworks provide near real-time processing, while batch frameworks offer increased processing throughput and fault-tolerance at the cost of higher

latency. Thus, streaming frameworks are more suitable for applications that run continuously, processing data as it arrives. Examples of such commercial applications are trending topics on Twitter, music recommendation on Spotify, weather prediction on The Weather Channel, tracking data at LinkedIn and others [4, 5]. On the other hand, batch frameworks are a better fit for more intense processing on large volumes of already stored data. Examples of such applications are web indexing at Google, search optimization on Ebay, machine learning on Facebook and all forms of data science and machine learning applications [6, 7].

None of the existing data processing systems is a perfect fit for all applications types, code developing styles, and cluster configurations. Due to their diverse design goals and maturity levels, processing frameworks are different in significant ways. They differ in the programming APIs and languages they offer, the community support and echosystem of add-on processing frameworks they provide, but also in their performance, scalability, and fault-tolerance levels. For this reason, users are increasingly tempted to use multiple frameworks to operate on the same or different data, each implementing a different aspect of their analysis needs.

To allow access to a common file-system (e.g., HDFS [8]), enable resource sharing, and support diverse processing requirements, these processing frameworks are often deployed together (multi-analytics) under the control of a resource manager. Multi-analytics deployments benefit both developers and cluster administrators. Developers are able to select the framework that fits best to their application requirements and coding preferences or "mix and match" frameworks for different parts of the data processing pipeline without the need to replicate data between different clusters. On the other hand, system administrators must monitor one, more compact, cluster, reduce their costs by sharing cluster resources among different frameworks, and take advantage of the scalability and robustness these systems offer.

Despite the flexibility, versatility, and cost benefits of multi-analytics deployments and the potential they bring for processing the vast amounts of IoT data in a robust way, there are performance and cooperation issues related to their combined use that remain unaddressed. First and foremost, big data analytics systems, in the IoT "era", often must operate in clusters where resources (memory and cpu) are limited, unlike the resource-rich environments that were traditionally hosting them. IoT analytics are increasingly offloaded to "edge computing" systems, deployed near where the data is collected, to provide low-latency actuation and decision support. However, big-data frameworks like Hadoop, Spark, and Storm and resource managers like YARN [9] and Mesos [10], were originally designed for very large-scale clusters to address the processing needs of tech-giants like Google, Yahoo, Facebook, Twitter etc. Under these large deployments, where resources are usually abundant, these frameworks have proven their robustness; their behavior and performance is well known and tested with a variety of diverse applications when they run in large, dedicated clusters. Yet, they are not thoroughly evaluated when they run together under resource-constrained cluster or cloud settings, a common use case in IoT deployments.

Secondly, existing resource sharing allocators are not sufficient to support deadline-driven workloads in resource-constrained, multi-analytic cluster settings. Such workloads and settings are common in IoT deployments (e.g. edge computing), where low-latency actuation (an application completing before a deadline) is necessary. Popular fair-share allocators [11, 12] do not support deadlines but instead focus on preserving fair-share. Recent research efforts that attempt to solve this problem, are not suitable for multi-analytics and resource constrained environments. Most of them are framework-specific [13, 14, 15, 16] and therefore cannot be deployed over a resource manager in multi-analytic settings. Others, require extra cluster resources to sample, simulate or extensively monitor application execution and build runtime models. Thus, they im-

pose significant cluster overhead which is not acceptable when resources are already constrained. Lastly, some of these solutions can only be effective in clusters with a great number of repetitive jobs [17, 18, 19]. Therefore they cannot support ad-hoc queries and they cannot be effective in clusters with a small number of job repetitions or large execution time dispersion (which are common in production clusters [20, 21]).

Thirdly, existing fair-share allocators are not designed for clusters with significant congestion. Consequently, their allocation is greedy and lacks admission control to prevent fair-share violations. Moreover, current fair-share allocators deployed on YARN and Mesos do not provide detection and correction mechanisms to restore fairness in case violations occur (violations can occur either randomly or due to misbehaving frameworks). Thus, in resource-constrained clusters, where congestions are frequent, the behavior of fair-share policies is uncertain. However, when fair-share is the desired policy, it is important to provide the necessary mechanisms that reliable preserve it under any cluster condition.

To address these issues and to take advantage of the processing power and fault-tolerant capabilities in resource-constrained clusters with deadline-driven, multi-analytics workloads, we propose and explore the following thesis question.

Can we achieve allocation fairness and performance objectives in resource constrained clusters for multi-analytics, through predictive resource allocation and admission control?

In order to answer the above question we:

- Investigate and characterize the performance and behavior of batch and stream processing systems in multi-tenant, resource constrained settings. The deployed systems should be widely used, open-source frameworks. Multiple processing engines should operate combined under the management of a resource-negotiation

framework responsible to fairly distribute the available cluster resources among them.

- Design and empirically evaluate an admission control strategy for resource managers that manage deadline-driven workloads in resource-constrained cluster settings. Compare the performance of the suggested strategy against existing fair-share policies in terms of satisfied deadlines and useful work under different deadline types.

- Investigate the efficacy of fair-share schedulers in resource-constrained cluster settings. Design and empirically evaluate a resource allocator that uses deadline information and historical workload analysis to adapt its allocations on the changing cluster conditions and efficiently use resources to achieve fairness and satisfy job deadlines.

To understand how popular analytics frameworks behave and interfere under resource constrained settings, we investigate the performance and behavior of distributed batch and stream processing systems that share resource constrained, private clouds managed by a resource manager. We examine how these systems interfere with each other and the resource manager, to evaluate the effect on systems performance, overhead, and fair resource sharing. Our experimental results show that in constrained environments, there is significant performance interference that manifests in multiple ways. First, the resource manager fails to always preserve fairness. Second, application performance is highly variable because it depends on the way the resource manager shares resources. Third, resource allocation among tenants with different scheduling designs, can lead to a form of deadlock and underutilization of system resources.

To meet deadlines and utilize the cluster productively in resource constrained multi-analytic settings, we design and built a resource allocator with admission control. Our

approach is framework-agnostic and as such, suitable for resource managers for big data multi-analytics systems, such as Apache Mesos and YARN. It builds a cluster-wide model based on historical job runs and deadlines to estimate the minimum amount of resources an admitted job needs to meet its deadline. If these resources are not available, it drops the job to avoid resource waste in jobs with infeasible deadlines. Our experiments with real-world traces show that our approach meets significantly more deadlines than existing fair-share approaches and eliminates resource-waste in resource constrained settings.

Finally, to preserve fairness and meet job deadlines under variable resource-constrained resources, we design an adaptive and fair allocation mechanism with admission control. Our approach uses deadline information and historical job execution logs and adapts to changing cluster conditions, to assign enough resources for each job to meet its deadline "just-in-time". The results, from trace-based simulation on large production YARN workloads, show that, in resource-constrained settings, our approach outperforms the existing fair-share allocation policy implemented on resource-managers like YARN and Mesos in terms of fairness, deadline satisfaction, and efficient resource usage.

Our contributions push the state of the art in resource allocation for cluster managers that distribute the resources of resource-constrained clusters across multiple big-data analytic engines. We show that existing allocation policies allow interference between frameworks that manifests in violation of fairshair, performance variability, starvation, and deadlocks. Moreover, our work introduces a novel allocation mechanism to satisfy deadlines in a framework-agnostic way through cluster-wide resource requirements prediction and admission control, suitable for integration with existing open-source cluster managers. We also show that our approach adapts to changing cluster resources to meet deadlines, preserve fairness, and utilize the cluster more efficiently compared to existing fair-share allocators. Lastly, the resource allocators we present do not waste precious cluster resources and do not require repetitive job patterns. Thus, they are suitable for

resource-limited IoT analytics clusters with generic processing workloads.

Our research solutions are directly applicable on existing open-source, widely used resource managers, like Mesos and Yarn, without requiring costly modifications of the resource manager or hard to maintain integrations with the processing engines. The resource allocation mechanisms we suggest operate on the resource-manager level and they do not require knowledge of the internal structures used by the processing engines that run on top. Thus, they can evolve separately following the evolution of the resource manager (e.g., Mesos), without the requirement to remain combatible with the multiple, diverse, and fast evolving frameworks (e.g., Spark, Hadoop) that run on top.

Chapter 2

Background

2.1 Distributed Big Data Analytics Systems

Big dala analytics is the collection of software applications and system frameworks that enable developers, statisticians, data-scientists and all form of practitioners to analyze and make sense out of the big volumes of collected data. Big data analytics applications are numerous, covering a wide span of industry sectors in health-care, transportations, banking, social-media, e-commerce and others [6, 7, 4, 5], with specialized frameworks and environments developed for different scientific disciplines like machine-learning and neural networks, graph mining, and statistics [22, 23, 24]. On the heart of this ecosystem is its infrastructure; distributed, fault-tolerant, and scalable data management and processing frameworks geared to provide a suitable environment to run big data applications.

We can classify big data processing frameworks based on the kind of data they handle in two categories: Batch processing frameworks (e.g. Hadoop), which deal with large, bounded and persistent data, and Stream processing (or streaming) frameworks (e.g. Storm), which compute over smaller portions of data as it enters the system. Even

though we use streaming frameworks, as a component of a realistic cluster deployment, to evaluate the interference of big data frameworks [25], in this work we focus on the performance, and resource allocation mechanisms to preserve fair-share and meet deadlines with batch processing frameworks. (After all, streaming applications run continuously to process incoming data and could not be bounded by a deadline.)

Each batch framework is optimized for different application types and development styles and code developed for it is not portable to other frameworks. Processing frameworks might differ in their programming models and offered APIs, fault-tolerance guaranties, performance and resource utilization, but also in terms of their community support, and available toolsets that run on top of them. Apache Hadoop was the first open-source implementation of MapReduce, a programing model designed at Google to deal with the unprecedented scale require to index web pages. [1] Hadoop follows a functional programming model exposing a simple programming API (map and reduce functions) that made it very attractive to developers. However, Hadoop has some limitations. Every job reads its input data, processes it, and then writes it back to HDFS. Consequently, for iterative algorithms, which want to read once, and iterate over the same data many times, this model poses significant I/O overheads. To improve or extend the capabilities of Hadoop, in recent years, we have seen an explosion of new data processing frameworks. Specifically, new frameworks as Spark, Flink, Naiad [26, 27, 28] take advantage of the increased memory capabilities of modern computer clusters and perform in-memory data processing, which makes them faster for the iterative processing required in machine learning and graph analytics workloads.

[1] "Hadoop" is often used as a synonym to "MapReduce", even though "Hadoop" has evolved to a much wider ecosystem that includes the "Hadoop Distributed File System (HDFS)", a resource manager (YARN), the MapReduce API, and many other components that provide the means for fault-tolerance, scalability and multi-tenancy. Throughout this work we use "Hadoop" to refer to the MapReduce programming model and "Hadoop ecosystem" when we refer to all the different components and layers of a typical Hadoop deployment.

2.2 Multi-Analytics

To give developers the opportunity to select the most suitable framework for their particular use-case, modern big data deployments host multiple frameworks on the same cluster. All the major big data providers [29, 30, 31] include both Hadoop and Spark (among others) on their software stack distributions. Moreover, recent scientific research focuses on the development of workflow managers to support multiple processing frameworks and automatically choose the best for the particular task at hand [32, 33, 34]. Thus, developers can choose either manually or automatically the framework that fits best to their programming preferences and application requirements or even combine frameworks for different steps of their data pipeline.

Despite their benefits, multi-analytics deployments are more complicated than standalone configurations. Each of the participating frameworks needs isolation for performance, reliability, and security reasons. A possible way to provide this isolation is to deploy each framework on a different subset of virtual or physical machines (i.e. statically split the cluster hosts among frameworks), and/or schedule their workflows in a way that they do not cause interference to each other. If common data access is required to avoid costly data transfers, then a distributed file system (e.g. HDFS) can run underneath. However, since the processing power of the cluster is statically divided between frameworks, while the file system is shared, data locality would be compromised. That is, the frameworks would not be able to schedule processing on the same node the data is located, causing costly data transfers of the network and increased latency. Moreover, in the lack of a central controlling entity, such a cluster would be very hard to monitor, manage, or modify (i.e. add/remove nodes or processing frameworks). Therefore, while providing this isolation manually is possible, it over-complicates the deployment procedure, is prune to errors and hard to manage, and it defeats the goal of seamlessly

running frameworks and efficiently utilizing cluster resources through sharing. Instead, administrators share their clusters between the different frameworks by using a resource manager framework.

2.3 Resource Managers

In multi-analytic systems, a resource manager is responsible to provide isolation and share cluster resources across the different data processing frameworks. Resource managers (also called "resource negotiators" or "cluster managers") contribute in the wider adoption of big data frameworks as they allow better cluster utilization, lower infrastructure costs and enable developers to experiment with different frameworks without expensive data replication across clusters. They also provide a separation between resource management and task processing, an abstraction that leads to increased scalability and separation of responsibilities. As new data processing frameworks continue to emerge and no one fits all use-cases, resource managers will remain on the center of modern big-data deployments.

To allow better increased scalability, the most popular open-source resource managers, Apache YARN and Mesos, share the common goal of allocating resources across multiple frameworks operating on the cluster without interfering with the internal scheduling of each framework. They are responsible for the inter-job scheduling; allocate resources to the frameworks submitting jobs. In turn, each processing framework is responsible to deal with the intra-job scheduling, that is to optimize the execution of a job by distributing the allocated resources between tasks. Intra-job scheduling differs significantly across frameworks, as it depends on the specific programming model that each processing framework implements. For inter-job scheduling though, both Mesos and YARN usually implement some variation of fair-sharing policy in order to equally share resources across

running jobs.

Existing allocators for resource managers are deadline-agnostic. Instead, they are trying to distribute resources in an equal way across all frameworks, assuming that all jobs in the cluster are of the same importance or that system administrators are responsible for dealing with deadline-driven workloads. That is, in the presence of jobs with increased priority and specific deadlines, administrators should add cluster resources, statically separate their clusters, or requires users to reserve resources ahead of time. Such solutions, however, are impractical and inefficient for resource constrained clusters or incur a significant extra cost.

Our goal is to provide an allocator for resource managers suitable for meeting deadlines and preserving fairness in resource constrained clusters. We achieve this goal with a new allocator, system, and empirical evaluation that does not rely on the intra-job allocators of the various processing frameworks but is, instead, independent (and thus less complicated) from their internal scheduling and programming model. Moreover, we investigate how to do so for clusters under resource constraints (such as those commonly found in IoT edge systems), and in ways that meet workload deadlines (required for many IoT applications) and fairly share resources across frameworks in multi-analytic settings, without adding significant overhead.

2.4 Real time systems

Our work shares similarities with past work done on scheduling for real-time systems [35, 36, 37, 38]. Unlike, the recent need for meeting deadlines in big-data systems, the requirement of guaranteeing latencies and response times is an integral part of real-time systems. On real-time systems both soft, hard deadlines, and their combination have been examined depending on how safety-critical the system is. We are currently

focusing on hard deadlines but combinations of jobs with hard and soft deadlines or without deadlines at all, is a natural extension to our current work. Furthermore, most deadlines in real-time systems are hard, as most of the applications running are critical, while in big-data systems soft deadlines for less time-critical applications (e.g., ad-hoc queries) are common.

Similar to big-data systems, real-time workloads can have jobs with inter-dependencies to other jobs [39], include periodic [40] and aperiodic [41] jobs, and experience processors failures [42], transient overloads [43], or network partitioning [44]. Our work currently considers only independent jobs. Taking into consideration job inter-dependencies, when they are known, could further improve our results on the cost of increased scheduling complexity. Examining performance under node or network failures would also be valuable future work. Lastly, on our work we do not try to exploit job periodicity. Our current traces indicate that periodic jobs are not necessarily the majority in a workload and therefore we provide an allocation mechanism that is does not depend on that. However, in clusters with increased job periodicity, exploiting such additional information could be beneficial.

Scheduling policies for real-time systems are categorized in static and dynamic [35]. A static approach requires prior knowledge of the schedule and based on this knowledge is able to offline calculate the desired schedule. On the other hand, an online approach, takes into consideration job submissions and non-predicted events, which makes it more costly and also a debatable fit for reliably meeting deadlines. Our approach is dynamic in that we are not assuming a-priori knowledge of the schedule, a requirement that would not be realistic for big-data workloads that include a big number of ad-hoc queries. However, our per-job allocations do not change after admission. By predicting the arrivals of new jobs and the completion time of current running jobs, we could apply incremental allocations (an initial allocation could later grow larger to meet the estimated requirements that

were not feasible to be satisfied on admission time). However, fully dynamic allocations (adapting both to higher or lower allocations based on current job progress) would require extensive monitoring, framework specific knowledge, and pre-emption capabilities. Pre-emption is a hard future to implement in big-data systems, while frameworks-specific knowledge and monitoring are not a good fit for the resource-constrained, multi-analytics settings we consider in this work.

Lastly, some important difference between real-time systems and distributed big-data systems are data locality, resource homogeneity, and communication overheads. In big-data systems data-locality can significantly influence performance and thus schedulers try to assign to jobs the resources that are closer to the data, while in real-time systems locality is not of similar importance. Similarly, in big-data systems resources are usually heterogeneous and communication can be a significant overhead, especially for smaller jobs. In our work, we retain Mesos logic, by offering resources and leaving the locality choice to the frameworks. We implicitly take into consideration the heterogeneity of resources and communication overheads by estimating runtimes based on observed performance across all the cluster resources and without using offline simulations or sampling that are by definition dependent on the communication overheads and processing capabilities of the specific set of resources used.

Chapter 3

Performance Interference of Multi-tenant, Big Data Frameworks in Resource Constrained Private Clouds

In this chapter, we investigate and characterize the performance and behavior of big/fast data systems in shared (multi-tenant), moderately resource constrained, private cloud settings. In such settings more resources (CPU, memory, local disk) cannot simply be added on-demand in exchange for an additional charge (as they can in a public cloud setting). While these technologies are typically designed for very large scale deployments such as those maintained by Google, Facebook, and Twitter they are also common and useful at smaller scales [45, 46, 47].

We empirically evaluate the use of Hadoop, Spark, and Storm frameworks in combination, with Mesos [48] to mediate resource demands and to manage sharing across these big data tenants. Our goal is to understand

- How these frameworks interfere with each other in terms of performance when they are deployed under resource pressure,
- How Mesos behaves when demand for resources exceeds resource availability, and
- The degree to which Mesos is able to achieve fair sharing using Dominant Resource Fairness (DRF) [12] in resource restricted cloud settings.

From our experiments and analyses, we find that even though Spark outperforms Hadoop when executed in isolation for a set of popular benchmarks, in a multi-tenant system, their performance varies significantly depending on their respective scheduling policies and the timing of Mesos resource offers. Moreover, for some combinations of frameworks, Mesos is unable to provide fair sharing of resources and/or avoid deadlocks. In addition, we quantify the framework startup overhead and the degree to which it affects short-running jobs.

In the next section, we provide background on Mesos and the analytics frameworks that we employ in this chapter. Section 3.2 details our experimental methodology, setup, and configuration and Section 3.3 presents our experimental results for multi-tenant, resource-restricted clouds. We present related work in Section 3.4 and our conclusions and future work in Section 3.6.

3.1 Resource Sharing on Mesos

In private cloud settings, where users must contend for a fixed set of data center resources, users commonly employ the same resources to execute multiple analytics systems and make the most out of the limited amount to which they have access. To understand how these frameworks interfere in such settings, we investigate the use of Mesos to manage them and to facilitate fair sharing. Mesos is a cluster manager that

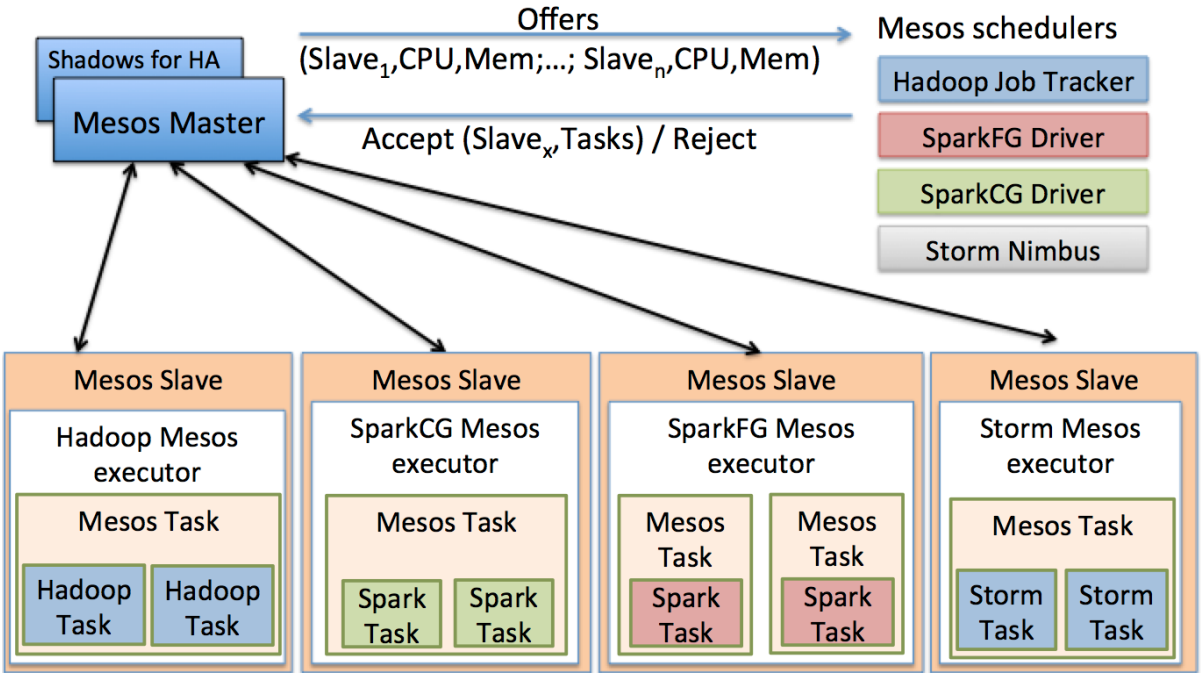


Figure 3.1: Mesos Architecture. Each distributed computing framework implements a Mesos scheduler which negotiates with the Mesos Master for Slave resource allocation, and a Mesos executor inside which framework tasks are spawned. Frameworks launch tasks in Coarse-Grained (CG) mode (1 Mesos task corresponds to 1+ framework tasks) or Fine-Grained (FG) mode (1 Mesos task corresponds to 1 framework task). Spark can use either CG or FG modes; Hadoop and Storm use CG.

can support a variety of distributed systems including Hadoop, Spark, Storm, Kafka, and others [48, 10]. The goal of our work is to investigate the performance implications associated with Mesos management of multi-tenancy for medium and small scale data analytics on private clouds.

We first overview the implementation of Mesos and its support for the analytics frameworks that we consider in this work: Hadoop, Spark, and Storm. Figure 3.1 provides a high level overview of the Mesos software architecture. Mesos provides two-level, offer-based, resource scheduling for frameworks. The Mesos Master is a daemon process that manages a distributed set of Mesos Slaves. The Master also makes offers containing available Slave resources (e.g. CPUs, memory) to registered frameworks. Frameworks accept or reject offers based on their own, local scheduling policies and control execution

of their own tasks on Mesos Slaves that correspond to the offers they accept.

When a framework accepts an offer, it passes a description of its tasks and the resources it will consume to the Mesos Master. The Master (acting as a single contact point for all framework schedulers) passes task descriptions to the Mesos Slaves. Resources are allocated on the selected Slaves via a Linux container (the Mesos executor). Offers correspond to generic Mesos tasks, each of which consumes the CPU and memory allocation specified in the offer. Each framework uses a Mesos Task to launch one or more framework-specific tasks, which use the resources in the accepted offer to execute an analytics application.

Each framework can choose to employ a single Mesos task for each framework task, or use a single Mesos task to run multiple framework tasks. We will refer to the former as "fine-grained mode" (FG mode) and the later as "coarse-grained mode" (CG mode). CG mode amortizes the cost of starting a Mesos Task across multiple framework tasks. FG mode facilitates finer-grained sharing of physical resources.

The Mesos Master is configured so that it executes on its own physical node and with high availability via shadow Masters. The Master makes offers to frameworks using a pluggable resource allocation policy (e.g. fair sharing, priority, or other). The default policy is Dominant Resource Fairness (DRF) [12]. DRF attempts to fairly allocate combinations of resources by prioritizing the framework with the minimum *dominant share* of resources.

The dominant resource of a framework is the resource for which the framework holds the largest fraction of the total amount of that resource in the system. For example, if a framework has been allocated 2 CPUs out of 10 and 512MB out of 1GB of memory, its dominant resource is memory ($2/10\ CPUs < 512/1024\ memory$). The dominant share of a framework is the fraction of the dominant resource that it has been allocated (512/1024 or 1/2 in this example). The Mesos Master makes offers to the framework

with the smallest dominant share of resources, which results in a fair share policy with a set of attractive properties (share guarantee, strategy-proofness, Pareto efficiency, and others) [12]. We employ the default DRF scheduler in Mesos for this work.

The framework implementations that we consider include the open source analytics systems Apache Hadoop [1], Apache Spark [2], and Apache Storm [3]. Hadoop implements the popular MapReduce programming model via a scalable, fault tolerant and, distributed batch system. Spark extends this model and system with an in-memory data-structure for Resilient Distributed Datasets (RDDs) [49]. Finally, Storm provides distributed, fault tolerant, real-time processing of streaming data. All frameworks leverage the Hadoop Distributed File System (HDFS) [8] for data persistence and durability. Each of these frameworks makes different design and implementation trade-offs (which result in different strengths and weaknesses), each is amenable to varying types of big data processing, analysis, and programming models, and each have had numerous applications written for them by developers [6, 7, 4]. For example, other studies show that Spark is much faster than Hadoop under normal conditions [50, 51], but that Hadoop has better fault-tolerance characteristics [50]. Moreover, Spark is using Resilient Distributed Datasets that make it a better choice compared to Hadoop for iterative algorithms, as it avoids repeated and costly reads and writes to/from HDFS, but this same mechanism is what slows it down compared to Hadoop when there is no data re-use on the workflow [32] or when data shuffling efficiency determines the performance [50].

In a Mesos deployment, each framework implements the Mesos scheduler interface and the Mesos executor interface. For Hadoop, the scheduler corresponds to the Hadoop *JobTracker* and the executor is a Hadoop *TaskTracker*. For Spark, the scheduler corresponds to the Spark *Driver* and there is a Spark extension that implements the executor for task management. For Storm, the scheduler is called *Nimbus* and the Mesos executors correspond to the Storm *Supervisors*. Storm Supervisors spawn one or more Storm

workers, each of which executes one or more application tasks as process threads.

Each framework creates one Mesos executor and one or more Mesos Tasks on each Slave in an accepted Mesos offer. In CG mode, frameworks release resources back to Mesos when all tasks complete or when the application is terminated. In FG mode, frameworks execute one application task per Mesos task. When a framework task completes, the framework scheduler releases the resources associated with the task back to Mesos. The framework then waits until it receives a new offer with sufficient resources from Mesos to execute its next application task. In our experiments we consider Spark, which provides both FG and CG modes as options, and Hadoop and Storm, which employ CG mode.

Throughout this chapter, we use the term "tenant" to refer to a framework (e.g. Spark, Hadoop, Storm, etc., that uses Mesos for cluster management. Thus, in single tenant scenarios, one framework submits a job and makes use of all the available cloud resources. Multi-tenancy refers to more than one framework running jobs at the same time, while sharing the same cluster resources managed by Mesos. Mesos applies its DRF policy to share the cluster resources and does not differentiate based on the number of frameworks that use the cluster.

Mesos supports Roles [52] to statically separate resources between multiple frameworks. There is no dynamic sharing of resources between different roles, but instead the total cluster capacity has to be divided across the frameworks. A static assignment of resources limits the peak capacity a cluster can support and wastes the resources when a frameworks is idle. These disadvantages are important in resource constrained environments, where the peak capacity and the available resources are already limited. Therefore, we did not make use of Mesos roles and instead investigate dynamic resource allocation sharing among frameworks.

3.2 Experimental Methodology

We next describe the experimental setup that we use for this work. We detail our hardware and software stack, overview our applications and data sets, and present the framework configurations that we consider.

We employ two, resource-constrained, Eucalyptus [53] private clouds, each with nine virtual servers (nodes). We use three nodes for Mesos Masters that run in high availability mode (similar to typical fault-tolerant settings of most real systems) and six for Mesos Slaves in each cloud. The Slave nodes on the first cloud (Eucalyptus v3.4.1), to which we refer to as *development*, each have 2x2.5GHz CPUs, 4GB of RAM, and 60GB disk space. The Slave nodes on the second, *production* cloud (Eucalyptus v4.1), have 4x3.3GHz CPUs, 8GB of RAM, and 60GB of SSD disk. Both clouds use Gigabit Ethernet switches.

Our nodes run Ubuntu v12.04 Linux with Java 1.7, Mesos 0.21.1 [54] which uses Linux containers by default for isolation, the CDH 5.1.2 MRv1 [55] Hadoop stack (HDFS, Zookeeper, MapReduce, etc.), Spark v1.2.1 [56], and Storm v0.9.2 [57]. We configure Mesos Masters (3), HDFS Namenodes, and Hadoop JobTrackers to run with High Availability via three Zookeeper nodes co-located with the Mesos Masters. HDFS uses a replication factor of three and 128MB block size. In addition, we found and fixed a number of bugs that prevented the Hadoop stack from executing in our environment. Our modifications are available at [58].

Our batch processing workloads and data sets come from the BigDataBench and the Mahout projects [59, 22]. We have made minor modifications to update the algorithms to have similar implementations across frameworks (e.g. when they read/write data, perform sorts, etc.). These modifications are available at [60]. In this work, we employ WordCount, Grep, and Naive Bayes applications for Hadoop and Spark and a WordCount streaming topology for Storm. WordCount computes the number of occurrences of each

		Development Cloud		Production Cloud	
		CPU	**MEM**	**CPU**	**MEM**
Available	**Slave**	2	2931	4	6784
	Total	12	17586	24	40704
Min Required	**Hadoop**	1	980	1	980
	Spark	2	896	2	896
	Storm	2	2000	2	2000
Max Used Per Slave	**Hadoop**	2	2816	4	5888
	Spark	2	896	4	896
	Storm	2	2000	4	4000
Max Used Total	**Hadoop**	12	16896	24	35328
	Spark	12	5376	24	5376
	Storm	6	6000	6	6000

Table 3.1: CPU and Memory availability, minimum framework requirements to run 1 Mesos Task and maximum utilized resources per slave and in total.

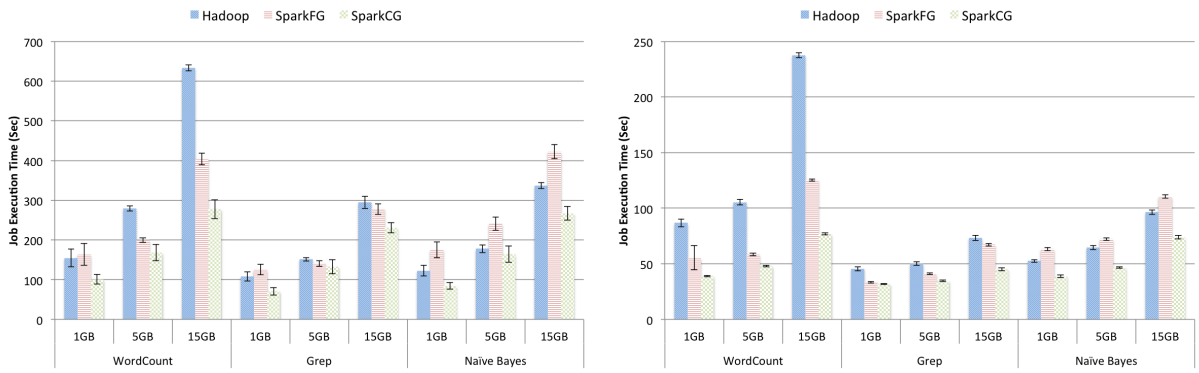
word in a given input data set, Grep produces a count of the number of times a specified string occurs in a given input data set, and Naive Bayes performs text classification using a trained model to classify sentences of an input data set into categories.

We execute each application 10 times after three warmup runs to eliminate variation due to dynamic compilation by the Java Virtual Machine and disk caching artifacts. We report the average and standard deviation of the 10 runs. We keep the data in place in HDFS across the system for all runs and frameworks to avoid variation due to changes in data locality. We measure performance and interrogate the behavior of the applications using a number of different tools including Ganglia [61], ifstat, iostat, and vmstat available in Linux, and log files available from the individual frameworks.

Table 3.1 shows the available resources in our two private cloud deployments, the minimum required resources that should be available on a slave for a framework to run at least one task on Mesos and, the maximum resources that can be utilized when the framework is the only tenant on the cloud. We configure the Hadoop TaskTracker with 0.5 CPUs and 512MB of memory and each slot with 0.5 CPUs, 768MB of memory,

and 1GB of disk space. We set the minimum and maximum map/reduce slots to 0 and 50, respectively. We configure Spark tasks to use 1 CPU and 512MB of memory, which also requires an additional 1 CPU and 384MB of memory for each Mesos executor container. We enable compression for event logs in Spark and use the default MEMORY ONLY caching policy. Finally, we configure Storm to use 1 CPU and 1GB memory for the Mesos executor (a Storm Supervisor) and 1 CPU and 1GB memory for each Storm worker.

This configuration allows Hadoop to run 3 and 7 tasks per Mesos executor for the development and production cloud, respectively. Hadoop spawns one Mesos executor per Mesos Slave and Hadoop tasks can be employed as either mapper or reducer slots. Spark in FG mode runs 1 Mesos/Spark task per executor on the development cloud and 3 Mesos/Spark tasks per executor on the production cloud. In CG mode, Spark allocates its resources to a single Mesos task per executor that runs all Spark tasks within it. In both modes, Spark runs one executor per Mesos Slave. We configure the Storm topology to use 3 workers. On the development cloud 1 Supervisor (Mesos executor) that runs 1 worker fits per slave and therefore 3 Slaves are needed in total. On the production cloud up to 3 workers can fit in the same Supervisor and therefore the Storm topology can be deployed in 1 Slave or distributed in multiple Supervisors across Slaves. We consider three different input sizes for the applications to test for small, medium and long running jobs. As the number of tasks per job is determined by the HDFS block size (which is 128MB), the 1GB input size corresponds to 8 tasks, the 5GB input size to 40 tasks and, the 15GB input size to 120 tasks.



(a) Development Cloud (b) Production Cloud

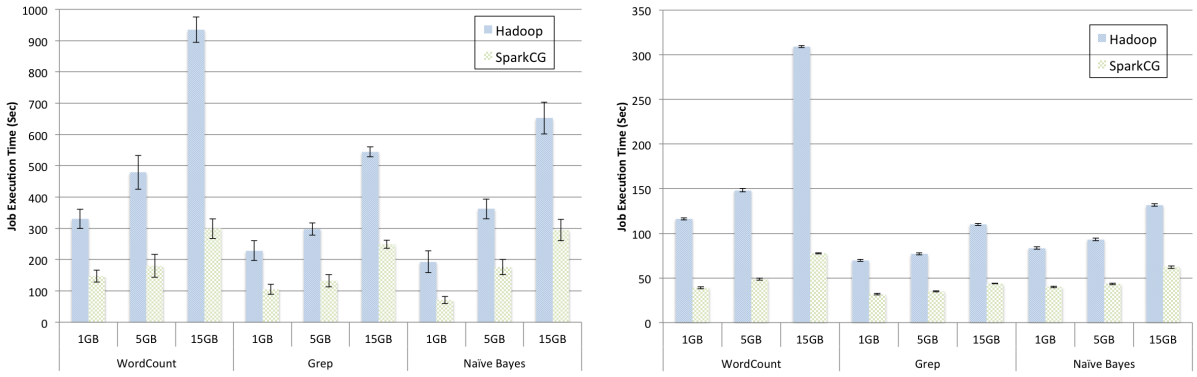
Figure 3.2: Single Tenant Performance: Benchmark execution time in seconds for Hadoop and Spark on Mesos for different input sizes. In this experiment, we execute each job using Hadoop, Spark in coarse grained mode (SparkCG), and Spark in fine grained mode (SparkFG), using the development cloud (Figure 3.2a) and production cloud (Figure 3.2b).

3.3 Results

For the first set of experiments, we use this experimental setup to measure the performance of Hadoop and Spark when they run in isolation (single tenancy) on our Mesos-managed private clouds. Throughout the remainder of this chapter, we refer to Spark when configured to use FG mode as *SparkFG* and when configured to use CG mode as *SparkCG*.

Figure 3.2 presents the execution time for the three applications for different data set sizes (1GB, 5GB, and 15GB) for the development cloud (left graph) and production cloud (right graph). These results serve as a reference for the performance of the applications when there is no resource contention (no sharing) across frameworks in our configuration.

The performance differences across frameworks are similar to those reported in other studies, in which Spark outperforms Hadoop (by more than 2x in our case) [50, 51]. One interesting aspect of this data is the performance difference between SparkCG and

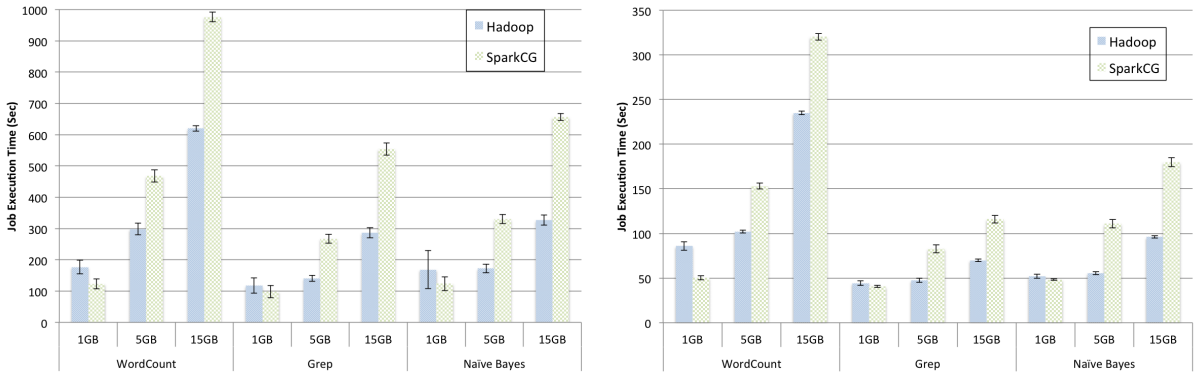


(a) Development Cloud (b) Production Cloud

Figure 3.3: Multi-tenant Performance: Benchmark execution time in seconds for Hadoop and SparkCG using different input sizes deployed on the development cloud (Figure 3.3a) and the production cloud (Figure 3.3b). In this setting, SparkCG receives offers from Mesos first because it is able to setup the application faster than Hadoop is able to.

SparkFG. SparkCG outperforms SparkFG in all cases and by up to 2x in some cases. The reason for this is that SparkFG starts a Mesos Task for each new Spark task to facilitate sharing. Because SparkFG is unable to amortize the overhead of starting Mesos Tasks across Spark tasks as is done for coarse grained frameworks, overall performance is significantly degraded. SparkCG outperforms SparkFG in all cases and Hadoop outperforms SparkFG in multiple cases. In particular, for the small 1GB input size on the development cluster, for which the increased latencies of Spark fine-grained correspond to a significant overhead on the total job runtimes, Spark performance is worse than Hadoop. This is also true for the Naive Bayes benchmark. In this case Spark first collects the classifier's model from HDFS in a separate stage with one running task on a sigle executor and it proceeds with staging the executors for the other tasks of the job only after completion of this stage. This delayed staging of executors on Mesos leads to slower runtimes for Spark Fine-Grained.

In the next experiment, we investigate the performance impact of multi-tenancy in



(a) Development Cloud (b) Production Cloud

Figure 3.4: Multi-tenant Performance: Benchmark execution time in seconds for Hadoop and SparkCG using different input sizes deployed on the development cloud (Figure 3.4a) and the production cloud (Figure 3.4b), when we delay Spark to ensure that Hadoop receives Mesos offers first.

a resource constrained setting. For this chapter, we execute the same application in Hadoop and SparkCG and start them together on Mesos. In this configuration, Hadoop and SparkCG share the available Mesos Slaves and access the same data sets stored on HDFS. Figure 3.3 shows the application execution time in seconds (using different input sizes) over Hadoop and SparkCG in this multi-tenant scenario. As in the previous set of results, SparkCG outperforms Hadoop for all benchmarks and input sizes.

3.3.1 Multi-tenant Performance

We observe in the logs from these experiments that SparkCG is able to setup its application faster than Hadoop is able to. As a result, SparkCG wins the race to acquire resources from Mesos first. To evaluate the impact of such sequencing, we next investigate what happens when Hadoop receives its offers from Mesos ahead of SparkCG. To enable this timing of offers, we delay the Spark job submission by 10 seconds. We present these results in Figure 3.4. In this case, SparkCG outperforms Hadoop for only the 1GB input

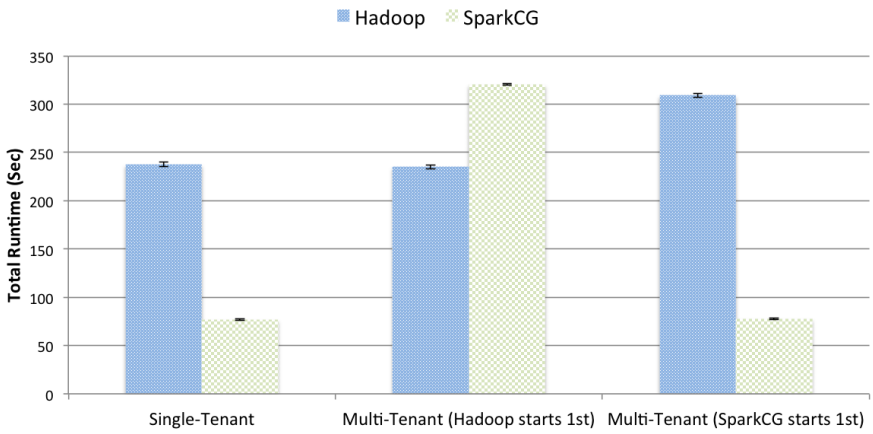


Figure 3.5: Performance Implications of Multi-tenancy and Mesos Offer Order: Hadoop and SparkCG. This graph shows WordCount execution time in seconds for input size 15GB using the production cloud (single-tenant, multi-tenant with Hadoop ahead of Spark, and multi-tenant with Spark ahead of Hadoop). The framework that receives Mesos offers first performs best.

size.

To understand this effect better, we summarize (i.e. we zoom in) the performance differences between Hadoop and SparkCG for different Mesos offer orders. Figure 3.5 shows execution time for WordCount and the 15GB input size using the production cloud. The first pair of bars shows the total time for the benchmark when each framework has sole access to the entire cluster (for reference from Figure 3.2b). The second pair of bars is the total time when Hadoop receives its resource offers from Mesos first. The third pair shows total time when SparkCG receives Mesos offers first (for reference from Figure 3.3b).

The data shows in this case that even though Spark is more than 160 seconds faster than Hadoop in single-tenant mode, it is more than 85 seconds *slower* than Hadoop when the Hadoop job starts ahead of the Spark job. Whichever framework starts first, executes with time similar to that of the single tenancy deployment.

This behavior results from the way that Mesos allocates resources. Mesos offers *all* of the available resources to the first framework that registers with it, since it is

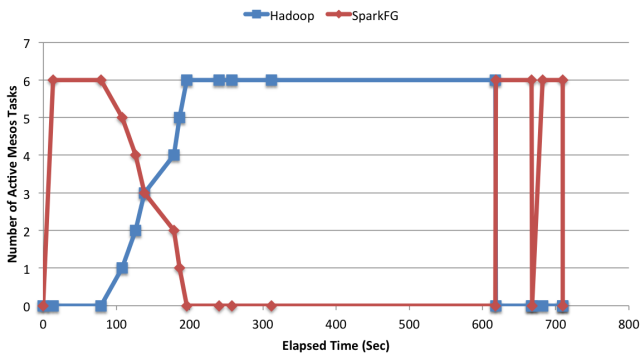
unable to know whether or not there will be a future framework to register. Mesos is incapable to change system-wide allocation when a new framework arrives, since it does not implement resource revocation. SparkCG and Hadoop will block all other frameworks until they complete execution of a job. In Hadoop, such starvation can extend beyond a single job, since Hadoop jobs are submitted on the same Hadoop JobTracker instance. That is, a Hadoop instance will retain Mesos resources until its job queue (potentially holding multiple jobs) empties.

These experiments show that when an application requires resources that exceed those available in the cloud (input sizes 5GB and above in our experiments), and when frameworks use CG mode, Mesos fails to share cloud resources fairly among multiple tenants. In such cases, Mesos serializes application execution limiting both parallelism and utilization significantly. Moreover, application performance in such cases becomes dependent upon framework registration order and as a result is highly variable and unpredictable.

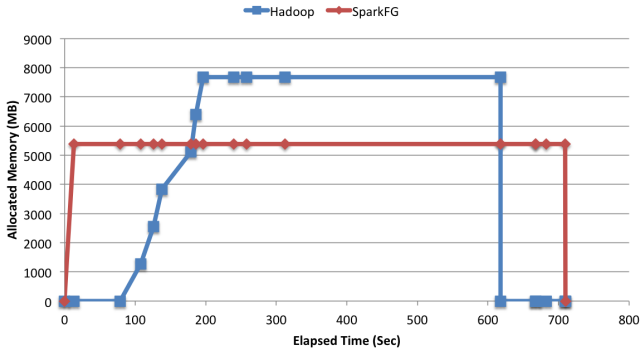
3.3.2 Fine-Grained Resource Sharing

In this section, we investigate the operation of the Mesos scheduler for frameworks that employ fine grained scheduling. For such frameworks (SparkFG in our study), the framework scheduler can release and acquire resources throughout the lifetime of an application.

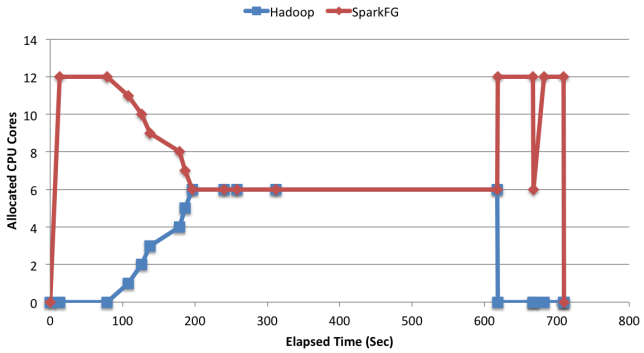
For these experiments, we measure the impact of interference between Hadoop and SparkFG. As in the previous section, we consider the case when Hadoop starts first and when SparkFG starts first. We present a representative subset of the results for clarity and brevity. Figure 3.7 shows the total execution time in seconds for WordCount and its 15GB input on the production cloud when we run Hadoop and SparkFG together and alter the Mesos offer order. As for Figure 3.5, we present three pairs of bars. The



(a) Number of active (staging or running) Mesos Tasks



(b) Memory allocation per framework



(c) CPU cores allocation per framework

Figure 3.6: Multi-tenancy and Resource Utilization: The timelines show active Mesos Tasks, memory, and CPU allocation in Mesos for the development cloud. Hadoop and Spark in FG mode compete for resources. Hadoop gradually takes over, running Tasks on its executors (Figure 3.6a), while memory (Figure 3.6b) and CPU cores (Figure 3.6c) previously assigned to Spark remain idle until Hadoop completes.

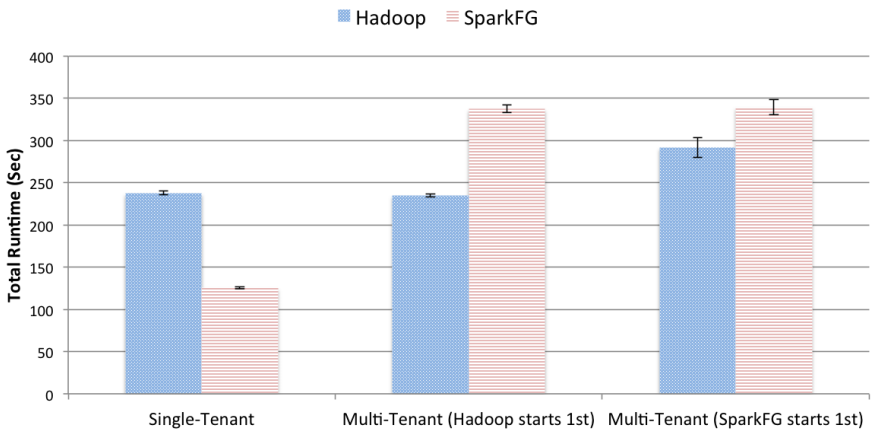


Figure 3.7: Performance Implications of Multi-tenancy and Mesos Offer Order: Hadoop and SparkFG. This graph shows WordCount execution time in seconds for input size 15GB using the production cloud (single-tenant, multi-tenant with Hadoop ahead of Spark, and multi-tenant with Spark ahead of Hadoop).

first, for reference, is the single-tenant performance. The second is the performance when Hadoop receives offers from Mesos ahead of SparkFG. For the third, SparkFG receives Mesos offers ahead of Hadoop.

As we expect, when Hadoop receives offers from Mesos first, it acquires all of the available resources, blocks SparkFG from executing, and outperforms SparkFG. Similarly, when SparkFG receives its offers ahead of Hadoop, we expect it to block Hadoop. However, from the performance comparison, this starvation does not occur. That is, Hadoop outperforms SparkFG (the far right pair of bars) even when SparkFG starts first and can acquire all of the available resources.

We further investigate this behavior in Figure 3.6. In this set of graphs, we present a timeline of multi-tenant activities over the lifetime of two WordCount/5GB applications (one over Hadoop, the other over SparkFG). In the top graph, we present the number of Mesos Tasks allocated by each framework. Mesos Tasks encapsulate the execution of one (SparkFG) or many (Hadoop) framework tasks. The middle graph shows the memory consumption by each framework and the bottom graph shows the CPU resources

consumed by each framework.

In this experiment, SparkFG receives first the offers from Mesos and acquires all the available resources of the cloud (all resources across the six Mesos Slaves are allocated to SparkFG). SparkFG uses these resources to execute the application and Hadoop is blocked waiting on SparkFG to finish. Because SparkFG employs a fine grained resource use policy, it releases the resources allocated to it for a framework task back to Mesos when each task completes. Doing so enables Mesos to employ its fair sharing resource allocation policy (DRF) and allocate these released resources to other frameworks (Hadoop in this case) – and the system achieves true multi-tenancy.

However, such sharing is short lived. As we can observe in the graphs, over time as SparkFG Mesos Tasks are released, they are allocated to Hadoop until only Hadoop is executing (SparkFG is eventually starved). The reason for this is that even though SparkFG releases its task resources back to Mesos, it does not release *all* of its resources back, in particular, it does not release the resources allocated to it for its Mesos executors (one per Mesos Slave).

In our configuration, SparkFG executors require 768MB of memory and 1CPU per Slave. Mesos DRF considers these resources part of the SparkFG dominant share and thus gives Hadoop preference until all resources in the system are once again consumed. This results in SparkFG holding onto memory and CPU (for its Mesos executors) that it is unable to use because there are insufficient resources for its tasks to execute but for which Mesos is charging under DRF. Thus, SparkFG induces a deadlock and all resources being held by SparkFG executors in the system are wasted (and system resources are underutilized until Hadoop completes and releases its resources).

In our experiments, we find that this scenario occurs for all but the shortest lived jobs (1GB input sizes). The 1GB jobs include only 8 tasks and so SparkFG will execute 6 out of its 8 task after getting all the resources on the first round of offers. Moreover, Hadoop

doesn't require all the Slaves to run 8 tasks for this job as explained on Section 3.2 leaving sufficient space to Spark to continue executing the remaining two tasks uninterrupted.

Deadlock in Mesos in resource constrained settings is not limited to the SparkFG scheduler. The fundamental reason behind this type of deadlock is a combination of (i) frameworks "hanging on" to resources and, (ii) the way Mesos accounts for resource use under its DRF policy. In particular, any framework scheduler that retains resources across tasks, e.g. to amortize the startup overhead of the support services (like Spark executors), will be charged for them by DRF, and thus may deadlock. Moreover, any Mesos system for which resource demand exceeds capacity can deadlock if there is at least one framework with a fine grained scheduler and at least one framework with a coarse grained scheduler.

3.3.3 Batch and Streaming Tenant Interference

We next evaluate the impact of performance interference in Mesos under resource constraints, for batch and streaming analytics frameworks. This combination of frameworks is increasingly common given the popularity of the *lambda architecture* [62] in which organizations combine batch processing to compute views from a constantly growing dataset and stream processing to compensate for the high latency between subsequent iterations of batch jobs and to complement the batch results with newly arrived unprocessed data [63, 64, 65].

We perform two types of experiments. In the first, we execute a streaming application using a Storm topology continuously, while we introduce batch applications. In the second, we submit batch and streaming jobs simultaneously to Mesos. Figure 3.8 illustrates the performance results for the former. We present execution time in seconds for the applications and input sizes using Hadoop, SparkFG, and SparkCG, when Storm

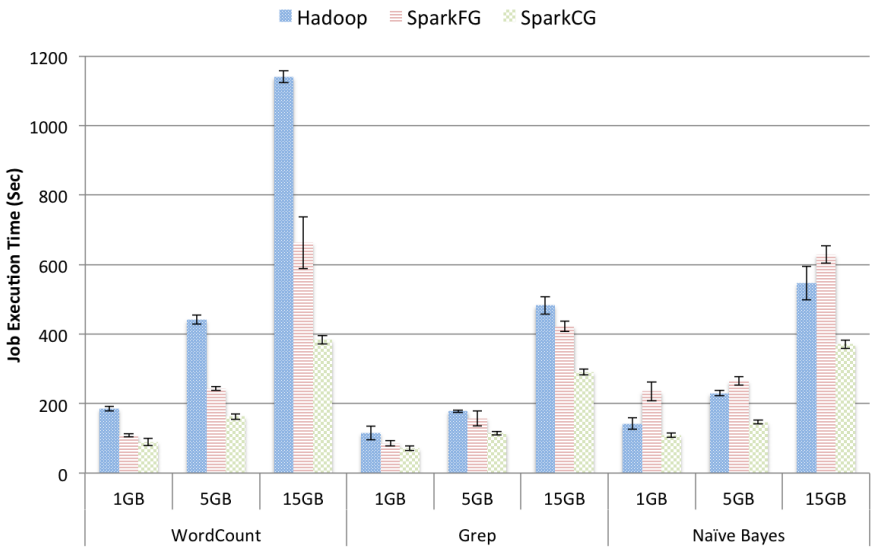


Figure 3.8: Hadoop and Spark performance on Mesos-managed cloud shared with Storm: Execution time in seconds for Hadoop and Spark applications in CG and FG mode, for different input sizes and benchmarks on the development cloud.

executes in the background. The results show that the performance degradation introduced by the Storm tenant varies between 25% to 80% across frameworks and inputs, and is insignificant for the 1GB input.

The reason for this variation is that Storm accepts offers from Mesos for three Mesos Slaves to run its job on the development cloud. This leaves three Slaves for Hadoop, SparkFG, and SparkCG to share. The degradation is limited because fewer Slaves impose less startup overhead on the framework executors per Slave. The overhead of staging new Mesos Tasks and spawning executors is so significant that it is not amortized by the additional parallelization that results from additional Mesos Slaves. We omit results for the production cloud for brevity. The results are similar but show less degradation (insignificant for 1GB, and 5% to 35% across frameworks and other inputs) due to the additional resources available in the production cloud.

Figure 3.9 shows the impact of interference from batch systems on Storm throughput in tuples per second (Results for latency are similar and we omit them for brevity). We

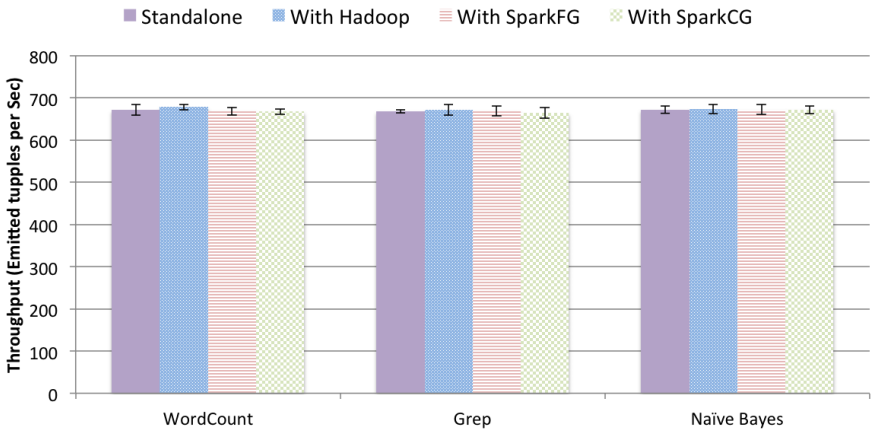


Figure 3.9: Impact of Multi-tenancy on Storm in Mesos: When Storm receives its offers from Mesos ahead of the batch frameworks, there is no interference impact on Storm, i.e. Storm throughput is the same in multi-tenant and single tenant deployments.

find that the interference is insignificant and Storm performance is the same as that when it executes in single-tenant mode, since Storm receives its offers and allocates the resources it needs ahead of the batch frameworks. When a coarse-grained batch system receives its resource offers from Mesos ahead of Storm, Storm execution is blocked until the batch system finishes. Figure 3.10 shows the timing diagram and this effect on Storm when executed with a SparkCG tenant in Mesos for the development cloud. Each line type shows a different stage of execution for each framework for each Mesos Task. Results with Hadoop and SparkFG are not shown for brevity. Hadoop has the exact same effect on Storm as SparkCG, while with SparkFG this depends on the cloud size. On the development cloud the released resources from SparkFG are not sufficient for Storm to deploy its executors and therefore Storm is blocked, while in the production cloud Storm will acquire some of the resources released by SparkFG as described previously, without however deadlocking SparkFG because Storm does not consume all of the cloud resources to run its tasks.

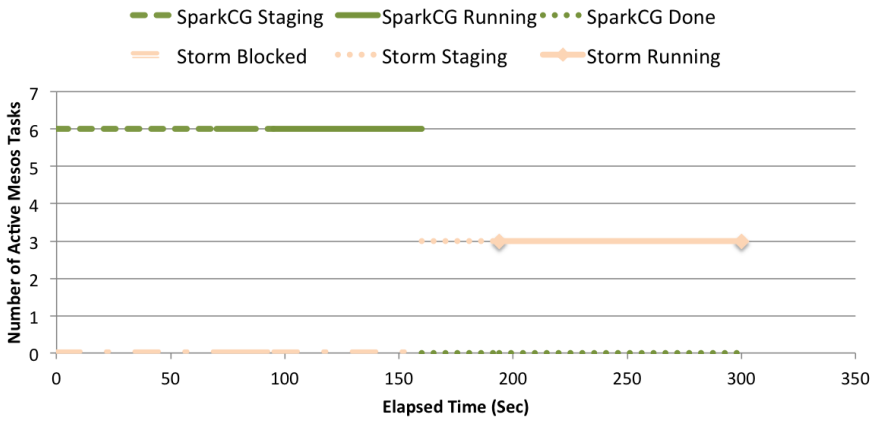
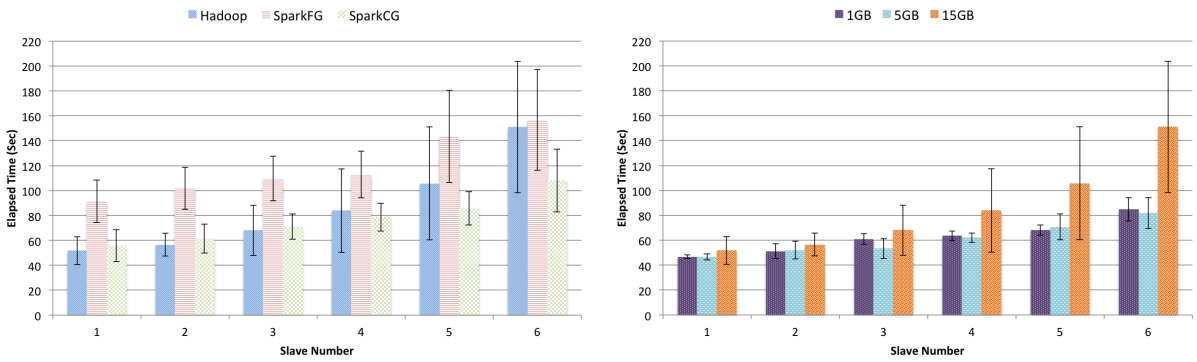


Figure 3.10: Multi-tenant Interference between Storm and SparkCG in Mesos, broken down by stage of execution: When SparkCG receives its Mesos offers first, Storm blocks until SparkCG finishes.

3.3.4 Startup Overhead of Mesos Tasks

We next investigate Mesos Task startup overhead for the batch frameworks under study. We define the startup delay of a Mesos Task as the elapsed time between when a framework issues the command to start running an application and when the Mesos Task appears as running in the Mesos user interface. As part of startup, the frameworks interact with Mesos via messaging and access HDFS to retrieve their executor code. This time includes that for setting up a Hadoop or Spark job, for launching the Mesos executor (and respective framework implementation, e.g. TaskTracker, Executor), and launching the first framework task.

Figure 3.11a shows the average startup time in seconds for each Mesos Slave across applications for Hadoop, SparkFG, and SparkFG when running the WordCount application with input size 15GB. Our experiments indicate that other applications perform similarly. The data shows that as new tasks are launched (each on a new Mesos Slave), the startup delay increases and each successive Slave takes longer to complete the startup process. Our measurements show that this increase is due to network and disk contention. Slaves



(a) Task startup delays across frameworks

(b) Task startup delays across input sizes

Figure 3.11: Task Startup Delays Across Frameworks (Figure 3.11a) and Input Sizes (Figure 3.11b) for the Development Cloud: The data shows the time in seconds required for each Mesos Task to finish its startup process.

that start earlier complete the startup process earlier and initiate application execution (execution of tasks). Task execution consumes significant network and disk resources (for HDFS access) which slows down the startup process of *later* Slaves. This interference grows with the size of application input as shown in Figure 3.11b. The graph shows the Mesos Task startup overhead in seconds for each Slave for different input sizes for WordCount over Hadoop (other frameworks exhibit similar behavior).

Our results show that startup overhead impacts the overall performance of applications and can significantly degrade the performance of short running jobs: 30% for Hadoop and 55% for SparkFG for the 1GB experiments. Given that short running jobs account for an increasingly large portion of big data workloads today [45, 46, 47], such overheads can cause significant under-utilization and widely varying application performance in constrained settings.

3.4 Related Work

On this section we present research done on resource managers, multi-analytics, performance comparison between Hadoop and Spark, Hadoop workload characterization, and performance startup overheads. We describe their impact to the state of the art and how our work differs or extends on these efforts.

Cluster managers like Mesos [48] and YARN [66] enable the sharing of cloud and cluster resources by multiple, data processing frameworks. YARN uses a classic resource request model in which each framework asks for the resources it needs to run its jobs. Mesos as described herein, implements an offer-based model in which frameworks can accept or reject offers for resources based on whether the offers satisfy the resource requirements of the application. Our work focuses on fair-sharing and deadlock issues that occur on Mesos due to lack of admission control and resource revocation. However, Mesos is not the only cluster manager that suffers from such problems. Other work [67] shows that, when the amount of required resources exceeds that which is available, deadlocks also occur on YARN.

Recently, new big data workflow managers that support multiple execution engines have emerged. Musketeer [32] dynamically maps a workflow description to a variety of execution engines, including Hadoop and Spark to select the best performing engine for the particular workflow. Similarly, [33] optimizes end-to-end data flows, by specializing and partitioning the original flow graph into sub flows that are executed over different engines. The advent of these higher-level managers calls for an increase in the combined use of data processing systems in the near future. Our work focuses on understanding system design limitations that will emerge under these new conditions.

The performance differences of MapReduce and Spark on very large clusters, with an emphasis on the architectural components that contribute to these differences is studied

in [50]. [68] evaluates the memory and time performance of Spark and MapReduce on Mesos, for the PageRank algorithm. The authors in [51] extend MPI to support Big Data jobs and compare performance and resource utilization of Hadoop and Spark. Ousterhout et al [69] suggest using blocked-time analysis to quantify performance bottlenecks in big data frameworks and apply it to analyze Spark's performance. In a recent work, Li et al [70] extend incremental Hadoop [71] to support low latency stream queries and compare the performance of their system to Spark streaming [72] and Storm. The key aspect that differentiates our work is our investigation and characterization of the performance of Hadoop, Spark and Storm applications, when run over Mesos cluster manager, in resource constrained and multi-tenant settings.

There are numerous studies that characterize the performance of MapReduce workloads. Many show that these workloads consist of many jobs (if not the majority) that have small input sizes and that have short execution times. Chen et al [45] observe that most jobs have input, shuffle and, output sizes in the MB to GB range and that 90% of the jobs have input datasets less than few GBs. Similarly, authors of [46] find that over 40% of jobs have less than 10 tasks, while about 50% of jobs have between 10 to 2000 tasks and, 80% of the jobs have duration less than 2 minutes. Lastly, the authors in [47] observe that small jobs dominate their workloads and that more than 90% of jobs touch less than 20GB of data, and their duration is less than 8 minutes.

Other researchers (e.g. [73, 74]) have shown the significant impact of startup overhead on MapReduce jobs that run on large cluster systems (hundreds to thousands of nodes). Our work differs from this past work in that we investigate the performance impact of multi-tenant interference on short running applications and analyze the overhead of job startup under moderate resource constraints. Such scenarios are increasingly common yet are not those for which large scale analytics systems were originally designed, warranting further study.

3.5 Key Insights

In this chapter, we use three different applications to expose the challenges to fair sharing in multi-tenant, resource constrained cluster settings. Regardless of application, the root cause of framework interference is how resources are allocated and shared. The applications we include are those used in previous performance studies [50, 51, 32]. WordCount and Grep are core components in text processing applications, and Naive Bayes Classification is a popular algorithm used in social network and e-commerce analytics.

3.5.1 Interference under Multi-tenancy

To study multi-tenancy, we consider two and three tenant scenarios. Our experience has been that these scenarios capture much of the interference behavior representative of higher numbers of tenants. The reason for this is fundamental to Mesos resource allocation. The framework that submits a job first, receives Mesos offers and therefore acquires all available resources. If the framework is coarse-grained it will keep these resources and block the other frameworks until it completes its job. If the first job that arrives is from a fine-grained framework, then the fine-grained framework will release the resources related to its tasks while keeping the resources related to its executors regardless of the number of tenants in the system.

3.5.2 Performance Interference and Possible Solutions

Performance interference occurs when the available cluster resources is less than the peak demand of the jobs running on the cluster which can occur on clusters of any size, but becomes the common case when resources are constrained. Static allocation of resources in constrained settings only exacerbates the problem. To reduce the impact of framework interference, Mesos must perform dynamic resource allocation and be extended to provide

either intelligent admission control, a resource revocation capability, or both, to avoid the performance degradations revealed in this chapter.

In particular, we can extend the resource manager to identify problematic behavior and revoke resources when fair sharing is violated. This requires the implementation of a pre-emption mechanism in each framework in order to avoid costly re-computations. Such a solution, however, will degrade performance of the framework from which the resources are revoked, but would enable a more predictable behavior of all the frameworks and guarantee continue progress of all the jobs.

Another way to address these issues is through job admission control. The resource manager can be extended to record historic information about job behavior or to support deadlines [13, 19, 20]. Mesos could then offer frameworks only the necessary resources required to meet a deadline to give the system more flexibility in achieving fair sharing. Deadline-driven admission control does not preclude all deadlock, but it can reduce its frequency.

We can also overcome framework interference in multi-tenant scenarios by forcing all frameworks to use fine-grained allocation in which they release all the resources related to their tasks and executors every time a task completes its execution. This way, even if the resource manager offers all the available resources to the framework that submits its job first, the framework will release resources upon task completion and these resources will be offered to the frameworks waiting. Such a requirement comes with a performance penalty as the overheads of creating executors and new Mesos tasks are significant but has the potential to increase performance stability and fair sharing in multi-tenant scenarios.

3.6 Summary

In this chapter, we characterize the behavior of three "big data" analytics frameworks in shared settings for which computing resources (CPU and memory) are limited. Such settings are increasingly common in both public and private cloud systems in which cost and physical limitations constrain the number and size of resources that are made available to applications. In this chapter, we investigate the performance and behavior of distributed batch and stream processing systems that share resource constrained, private clouds managed by Mesos. We examine how these systems interfere with each other and Mesos, to evaluate the effect on systems performance, overhead, and fair resource sharing.

We find that in such settings, the absence of an effective resource revocation mechanism supported by Mesos and the corresponding data processing systems running on top of it, leads to violation of fair sharing. In addition, the naive allocation mechanism of Mesos benefits significantly the framework that submits its application first. As a result coarse-grained framework schedulers cause resource starvation for later tenants. Moreover, when systems (either batch or streaming) with different scheduling granularities (fine-grained or coarse-grained) co-exist on the same Mesos-managed cloud, resource underutilization and resource deadlocks can occur. Finally, the overhead introduced during application startup on Mesos affects all frameworks and significantly degrades the performance of short running applications.

Chapter 4

PYTHIA: Admission Control for Multi-Framework, Deadline-Driven, Big Data Workloads

In the previous chapter, we saw that resource allocators without preemption mechanism and admission control, cannot prevent or correct framework interference. Interference hinders fair-sharing and reliable application performance in resource-constrained multi-analytics clusters managed by resource negotiators. Improving the predictability of application performance is important to satisfy tight deadlines of IoT workloads and avoid wasting valuable computation time and resources from the already resource-limited IoT analytics clusters.

In this chapter, we investigate a simple, new admission control strategy for resource negotiators called Pythia – for deadline-driven workloads in resource-constrained settings. Such workloads represent an important class of big data applications [13, 75, 76], which are unfortunately not fully supported in multi-analytic settings. Recent advances optimize based on framework-specific characteristics [17, 14, 15], depend upon job repe-

tition [45, 77, 20] (which is not always available), use additional cloud resources to build offline performance and scalability models [20, 78, 19] or require complicated system extensions such as job pre-emption [79, 80]. Approaches that are framework specific (e.g. for Hadoop [17, 81, 14, 18, 82, 83, 84, 85] or Spark [15, 16] in isolation), cannot be used directly by resource negotiators. A lack of deadline support can lead to deadline misses when resources are constrained. Deadline misses disrupt big data pipeline processing, lead to inefficient resource utilization, increase the cost of cloud use, and lead to poor decision making (e.g. when based on stale information or analysis).

To address these limitations, we design Pythia to be framework-agnostic and to not rely on job repetition or pre-emption support. Instead, Pythia uses histories of job statistics (resource negotiator log information) to control job admission. Instead of allocating all or a fair-share of CPUs to a job, Pythia allocates the fraction of this number that it estimates will enable the job to complete by its deadline. By "slowing down" the job while still meeting its deadline, Pythia is able to save resources and meet deadlines of other jobs. Pythia does not admit infeasible jobs (those that are likely to miss their deadlines) and thus, jobs submissions "fail fast" without wasting cloud resources. Extant systems admit all jobs and then "react" when a deadline passes by terminating or pre-empting a running job. Pythia uses jobs performance statistics from completed jobs to compute/update this allocation fraction online.

We implement Pythia for Mesos and empirically evaluate it using trace-based simulation of different cloud capacities. We use two real-world, 3-month, Hadoop traces from a production YARN system (contributed to us by an industry partner) for our experimentation. We evaluate Pythia and popular fair-share allocators, in terms of deadlines missed and useful work completed versus an oracle. We also consider both fixed and random deadline assignments. We find that Pythia achieves performance similar to that of the oracle in terms of deadlines met and useful work done. In addition, we find

that PYTHIA is able to meet significantly more deadlines than the fair share allocators for the workloads, cluster sizes, and deadline assignments that we consider.

In the sections that follow, we first overview deadline-driven admission control for big data cloud workloads. We then present the design and implementation of PYTHIA and overview the experimental methodology that we employ to evaluate its efficacy. We then present an empirical evaluation of PYTHIA, related work, and our conclusions.

4.1 PYTHIA Design

PYTHIA is a deadline-aware resource allocator with admission control for resource negotiators that manage the execution of batch applications with deadlines, using resource-constrained, shared clouds. PYTHIA employs a black-box, framework-agnostic technique, to estimate the minimum number of CPUs (parallelism) that a job requires to meet its deadline. To enable this, PYTHIA employs a simple yet effective approach that does not require job clustering, modeling, sampling, or complex simulation.

4.1.1 Admission Control

To design PYTHIA, we make a number of simplifying assumptions. First, we assume that the number of tasks for a job (the division of its input size and the HDFS block size) is the maximum parallelization possible for the job. We refer to this number as the `requestedTasks` for the job. To determine how many CPUs to allocate to a new job, PYTHIA uses performance data from past jobs. PYTHIA analyzes each job when it completes and uses this information to estimate the number of CPUs that the job *would have needed* to have finished by its deadline (`deadlineCPUs`). PYTHIA also assumes that there is perfect parallelism (speedup per CPU).

PYTHIA tracks the maximal global fraction, which we call `CPUFrac`, which is `deadlineCPUs`

over `requestedTasks` for jobs that complete. It multiplies this fraction by the number of tasks requested by frameworks for new jobs, to compute the number of CPUs to allocate to each job. Moreover, PYTHIA is *proactive* in that it prevents infeasible jobs (jobs likely to miss their deadlines) from ever entering the system and consuming resources wastefully. In this way, PYTHIA attempts to maximize the number of jobs that meet their deadline even under severe resource constraints (i.e. cloud capacity).

4.1.2 Algorithm

In this section we describe in detail PYTHIA's monitoring, admission control and resource allocation mechanisms. Algorithm 2 summarizes the monitoring mechanism, while Algorithm 1 provides the pseudocode for the most important parts of resource allocation and admission control. These, include the estimation of minimum required resources for a job to meet its deadline, a priority policy to order jobs, and the creation of resource allocation offers.

To bootstrap the system, PYTHIA sets `CPUFrac` to -1 and admits all jobs regardless of deadline; it allocates `requestedTasks` CPUs to the job. For any job for which there are insufficient resources for the allocation, PYTHIA allocates the number of CPUs available. When a job completes (either by meeting or exceeding its deadline), PYTHIA invokes the pseudocode function `TRACK_JOB` shown in Alg. 2 to potentially update `CPUFrac`.

`TRACK_JOB` calculates the minimum number of CPUs required (`deadlineCPUs`) if the job were to complete by its deadline, using its execution profile (available from the resource negotiator logs). Line 2 in the function is derived from the equality:

$$numCPUsAllocd * jobET = deadlineCPUs * deadline$$

On the left is the actual computation time by individual tasks, which we call `compTime`

Algorithm 1 Admission Control and Resource Allocation

```
1: function ADMISSION_CONTROL(RequesterJob)
2:     for all j ∈ SubmittedJobs do
3:         Feasible = True, TTD = Deadline − ElapsedTime
4:         reqCpus = ESTIMATE_REQ(j, TTD)
5:         if reqCpus > min(taskCount, capacity) then
6:             Feasible = False
7:         end if
8:         if Share(j) < reqCpus then
9:             if Feasible == True then
10:                priority = reqCpus/TTD, ADD_TO_HEAP(priority, j)
11:            else
12:                DROP_JOB(j)
13:            end if
14:        end if
15:    end for
16:    allocations = ALLOC_RESOURCES(heap)
17:    if RequesterJob ∉ allocations then
18:        Add RequesterJob to queue
19:    end if
20: end function

21: function ESTIMATE_REQ(Job)
22:    maxCpus = min(tasks, capacity), reqCpus = maxCpus
23:    if CompletedJobs > 1 then
24:        fraction = (deadline/(deadline − queue)) ∗ CPUFraq
25:        reqCpus = max(ceil(fraction ∗ maxCpus), 1)
26:    end if
27:    return reqCpus
28: end function

29: function ALLOC_RESOURCES(heap)
30:    offers = CREATE_OFFERS(heap)
31:    allocations = SEND_OFFERS(offers)
32:    return allocations
33: end function

34: function CREATE_OFFERS(heap)
35:    while availableCpus > 0 and heap not empty do
36:        for all Job j ∈ heap do
37:            offer = min(request(j), availableCpus)
38:            if offer < request(j) then
39:                offer = 0
40:            else
41:                availableCpus− = offer
42:                offersDict[j] = offer
43:            end if
44:        end for
45:    end while
46:    return offersDict
47: end function
```

Algorithm 2 Job Completion Monitoring

```
1: function TRACK_JOB(compTime, requestedTasks, deadline)
2:     deadlineCPUs = compTime/deadline
3:     maxCPUs = min(requestedTasks, cloud_capacity)
4:     minReqRate = deadlineCPUs/maxCPUs
5:     if minReqRate > CPUFrac then
6:         CPUFrac = minReqRate
7:     end if
8: end function
```

in the algorithm. `numCPUsAllocd` is the number of CPUs that the job used during execution and `jobET` is its execution time without queuing delay. The right side of the equation is the total computation time consumed across tasks if the job had been assigned `deadlineCPUs`, given this execution profile (`compTime`). `deadline` is the time (in seconds) specified in the job submission. By dividing `compTime` by `deadline`, we extract `deadlineCPUs` for this job.

Next, Pythia divides `deadlineCPUs` by the maximum number of CPUs allocated to the job. The resulting `minReqRate` is a fraction of the maximum that Pythia could have assigned to the job and still have it meet its deadline. Pythia compares `deadlineCPUs` against the global `CPUFrac` and updates `CPUFrac` if the former is larger. Pythia then uses `CPUFrac` as a prediction for this fraction for future jobs. `CPUFrac` is always less than or equal to 1. The tighter the deadline, the more conservative (nearer to 1) Pythia's resource provisioning will be.

Once `CPUFrac` has been initialized (from -1), Pythia employs it for admission control. To do so, Pythia multiplies `CPUFrac` by the number of tasks requested in the job submission (rounding to the next largest integer value). It uses this value (or the maximum cloud capacity, whichever is smaller) as the number of CPUs to assign to the job for execution. If this number of CPUs is not available, Pythia enqueues the job. Pythia performs this process each time a job is submitted or completes. It also updates the deadlines for jobs in the queue (reducing each by the time that has passed since sub-

mission), recomputes the CPU allocation of each enqueued job using the current `CPUFrac` value,this work and drops any enqueued jobs with infeasible deadlines.

Pythia admits jobs from the queue based on a pluggable priority policy. We have considered various policies for Pythia and use deadline maximization for the results in this chapter. In this policy, Pythia gives priority to jobs with a small number of tasks and greatest time-to-deadline. However, all of the policies that we considered (including shortest time-to-deadline) perform similarly. Once Pythia has selected a job for submission, it allocates the CPUs to the job and admits it to the system for execution. Once a job enters the system, its CPU allocation does not change.

4.2 Experimental Methodology

In this section, we describe the experimental methodology that we use to evaluate Pythia. We overview the trace-based simulation system, define our performance metrics, and present the deadline types that we consider.

We implement a discrete event simulator to empirically evaluate clouds under different resource constraints (number of instance cores available). Our system is based on Simpy [86] and replicates the execution behavior of production traces of big data workloads (cf Section 4.3). It supports multiple allocators that implement different policies for resource allocation and admission control. In this work, we consider:

`NoDrop FS`: This allocator employs a fair sharing policy [12, 87, 88, 89, 90] without admission control. Its behavior is similar to that of the default allocator in Mesos and YARN. Its goal is to share the cluster resources fairly among frameworks and it is unaware of deadlines. Therefore, it never drops jobs even when there are insufficient resources to meet the deadlines or if the deadlines have passed.

`Reactive FS`: This allocator extends NoDrop FS and has no admission control. When a

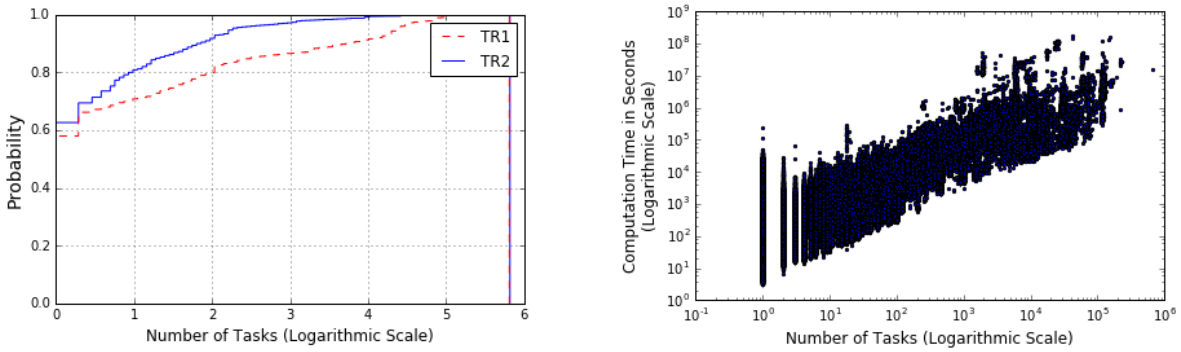
job exceeds its deadline, this allocator "reacts" and terminates the job, freeing resources in the cloud for use by other jobs.

`Oracle`: This allocator allocates the minimum number of resources that a job requires to meet its deadline. If sufficient resources are unavailable, the Oracle queues the job until the resources become available or until its deadline has passed (or is no longer achievable). This allocator is an oracle in the sense that it has future knowledge of actual execution time of jobs and thus will not admit jobs that will not meet their deadlines. However, it does not have a global view (i.e. ideal schedule). For the queued jobs, Oracle gives priority to jobs with fewer required resources and longer time until the deadline.

PYTHIA: As described in Section 4.1, this allocator proactively drops, enqueues, or admits jobs submitted. It estimates the number of resources to allocate using the global *CPUFrac* which it computes from job performance histories. For the queued jobs, PYTHIA gives priority to jobs with fewer required resources and longer computation times. PYTHIA drops any jobs that are infeasible based on their deadlines.

Deadline Types. We evaluate the robustness of our approach by running experiments using different deadline types as is done in prior related work [91, 92, 17, 13, 20]. In particular, we assign deadlines that are multiples of the optimal execution time of a job (which we extract from our workload traces). We use two types of multiples: Fixed and variable. With fixed deadlines, we use a deadline of 2 times the optimal execution time as is done in [91, 92]. For variable deadlines, we compute deadline multiples by sampling distributions. We randomly pick a deadline between two possible values as is done in the Jockey study [20] from the sets with values $(1, 2)$ and $(2, 4)$, which we refer to as Jockey1x2x and Jockey2x4x. We also experiment with deadline multiples that are uniformly distributed in the intervals $[1, 3]$ and $[2, 4]$ as used in Aria [17, 13]; we refer to these experiments as Aria1x3x and Aria2x4x, respectively.

Evaluation Metrics. Our goal with PYTHIA is to maximize the number of satisfied



(a) CDFs of the number of tasks per job in TR1 and TR2 (b) Job computation time vs number of tasks in TR2

Figure 4.1: **Workload Characteristics:** Number of tasks per job and computation time relative to jobs size (in number of tasks). Small jobs are large in number but consume a very small proportion of trace computation time.

deadlines and to avoid wasting resources on infeasible jobs. We assess the quality of Pythia using *Satisfied Deadlines Ratio (SDR)* and *Productive Time Ratio (PTR)*. SDR is the fraction of the jobs that completed before their deadline to the total number of submitted jobs. For the set of all the submitted jobs $J1, J2, ..., Jn$, let m be the subset of successful jobs $J1, J2, ...Jm$. Then SDR is: $\frac{\sum_{i=1}^{m} Ji}{\sum_{j=1}^{n} Jj}$. PTR is the fraction of the total computation time (across tasks) spent for jobs that satisfied their deadlines over the total computation time of all the jobs in the trace, regardless of whether they completed their execution. For the set of all the submitted jobs $J1, J2, ..., Jn$ with corresponding runtimes $T1, T2, ..., Tn$ we consider the subset of m successful jobs $J1, J2, ...Jm$. PTR then is: $\frac{\sum_{i=1}^{m} Ti}{\sum_{j=1}^{n} Tj}$.

4.3 Workload Characterization

To evaluate Pythia, we use two 3-month traces from production Hadoop applications executing over different YARN system. The traces were recently donated to our

Trace	CPUs	Jobs	Comp. Time (Hours)	1-Task Pct	1-Task Time Pct
TR1	9345	159194	8585673	58%	0.1%
TR2	24721	1140064	13301659	62%	0.3%

Table 4.1: Summary of Traces. Columns are trace name, peak cloud capacity, total number of jobs, total computation time in hours, percentage of 1-task jobs, and percentage of 1-task job computation time.
research lab by an Industry partner. Each trace contains a job ID, job category, number of map and reduce tasks, map and reduce time (computation time across tasks), job runtime, among other data. We have no information about the scheduling policy or HDFS configuration used in each cluster. Thus we use and assume a minimum of one CPU per task and use this minimum to derive cloud capacity; we are considering sub-portions of CPUs (vcores) as part of future work. Pythia uses the number of map tasks (as `requestedTasks`) and map time (as `compTime`) from the traces for simplicity; map task count and time dominate reduce task count and time for most jobs in both traces.

Table 5.1 summarizes the job characteristics of each trace. The table shows the peak cloud capacities (maximum number of CPUs in use), the total number of jobs, the total computation time across all tasks in the jobs, the percentage of jobs that have only one task, and the percentage of computation time that single-task jobs consume across jobs. We refer to the trace with 159194 jobs as TR2 and the trace with 1140064 jobs as TR2. The peak observed capacity (total number of CPUs) for TR1 is 9345 and for TR2 is 24721.

The table also shows that even though there are many single-task jobs, they consume a small percentage of the total computation time in each trace. To understand this characteristic better, we present the cumulative distribution of number of tasks per job in Figure 5.1 on a logarithmic scale. Approximately 60% of the jobs have a single task

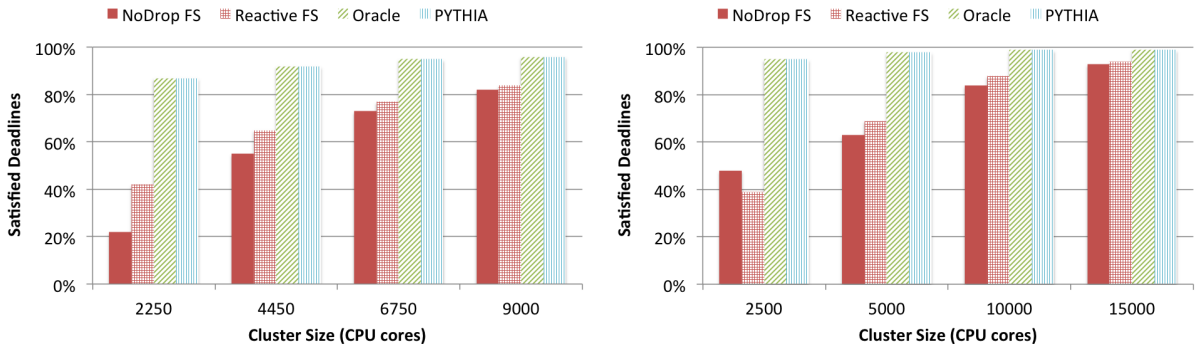
and 70-80% of the jobs have fewer than 10 tasks, across traces. Only 13% of the jobs in TR1 and 3% of the jobs in TR2 have more than 1000 tasks.

The right graph in the figure compares job execution time with the number of tasks per job (both axes are on a logarithmic scale) for the TR2 trace (TR1 exhibits a similar correlation). In both traces, 80% of the 1-task jobs and 60% of the 2-10 task jobs finish in fewer than 100 seconds. Their aggregate computation time is less than 1% of the total computation time of the trace. Jobs with more than 1000 tasks account for 98% and 94% of the total computation time for TR1 and TR2, respectively. Finally, job computation time varies significantly across jobs.

We have considered leveraging the job ID and number of map and reduce tasks to track repeated jobs, but find that for these real-world traces such jobs are small in number. In TR1, 18% of the jobs repeat more than once and 12% of the jobs repeat more than 30 times. In TR2, 25% of the jobs repeat more than once and 16% of the jobs repeat more than 30 times. Moreover, we observe high performance variation within each job class. Previous research has reported similar findings with production traces [20].

4.4 Empirical Evaluation

We evaluate Pythia using the production traces for different cloud capacities (number of CPUs) to evaluate its impact in both resource-constrained and resource-rich scenarios. Our cloud capacities range from 2250 and 15000 CPUs. We compare Pythia against different fair share schedulers, using two different deadline strategies, a random multiple (Jockey) and a uniform multiple (Aria) of the actual computation time, as described in Section 4.2. We have also implemented a prototype of Pythia as an allocation module [93] on Apache Mesos 0.27.2 which will make available on Github following publication.



(a) TR1 (b) TR2

Figure 4.2: Satisfied Deadline Ratio (SDR) with Fixed Deadlines for TR1 (left graph) and TR2 (right graph). All jobs have deadline multiples of 2.

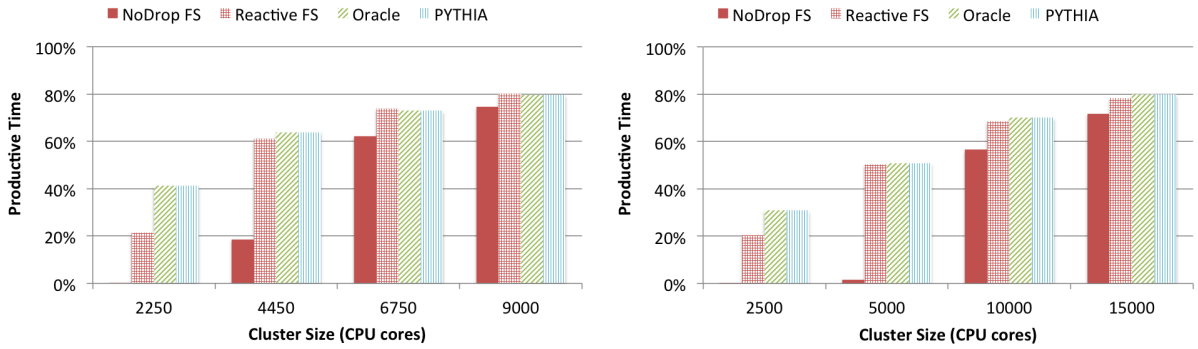
4.4.1 Fixed Deadlines

Figure 4.2 presents the deadline satisfaction ratio (SDR) for each allocator and multiple cloud capacities, when we employ deadlines that are 2 times the job runtime for TR1 (left graph) and TR2 (right graph). For both traces, PYTHIA meets significantly more deadlines than both of the fair share policies and performs similarly to the Oracle. Under tight resource constraints (the smallest cloud capacities), PYTHIA satisfies 295% more deadlines than NoDrop FS and and 143% more deadlines than Reactive FS.

As described previously, the Oracle does not have perfect information (i.e. it does not have a global optimal schedule) but it does know the actual job total computation time (`compTime`). Thus, it is able to assign the minimum number of CPUs to each job to satisfy its deadline. SDR for Oracle is not 100% because it must drop (refuse to admit) jobs for which there is insufficient capacity to meet their deadline.

We next evaluate the useful work (PTR) that each allocator enables. PTR is the fraction of total computation time across all jobs, that is due to jobs that are admitted and that meet their deadlines. Figure 4.3 shows these results.

For PTR, PYTHIA results are similar to those of the *Oracle* and PYTHIA outperforms



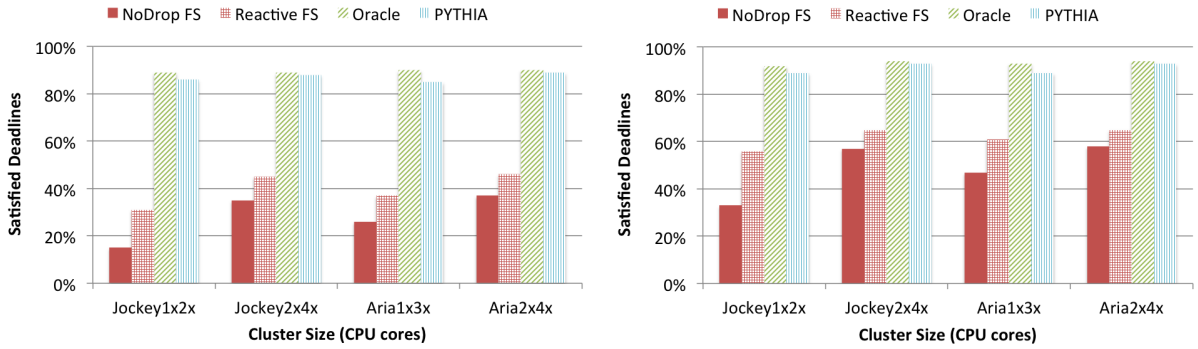
(a) TR1 (b) TR2

Figure 4.3: Productive Time Ratio (PTR) with Fixed Deadlines for TR1 (left graph) and TR2 (right graph). All jobs have deadline multiples of 2.

both fair share allocators across cloud capacities. In particular, PYTHIA facilitates up to 93% more productive time than to Reactive FS. With limited capacity, almost no useful work is achieved using NoDrop FS.

This pair of results shows that as resources become scarce, techniques without intelligent admission control for workloads with fixed deadlines lead increasingly to missed deadlines and wasted resources. This occurs because fair share allocators assign all or most CPUs (depending on their fair share fraction) required for the tasks requested. Because there is no deadline-awareness, these allocators do so regardless of whether or not the job is likely to meet its deadline. Jobs that miss their deadlines result in wasted (unproductive) work for the duration of the job for NoDrop FS and until the point the deadline is reached for Reactive FS.

In constrained clouds, the fair share allocators satisfy deadlines without contributing much to PTR. For example, *NoDrop FS* successfully completes approximately 20% of the jobs in the TR1. Yet, its PTR is near zero. In constrained settings, fewer resources are shared between jobs, leading to a smaller fair share per job. For larger jobs, this fair share is insufficient to meet their deadlines. For jobs with a small number of tasks, this smaller



(a) Cloud Capacity: 2250 CPU cores

(b) Cloud Capacity: 4500 CPU cores

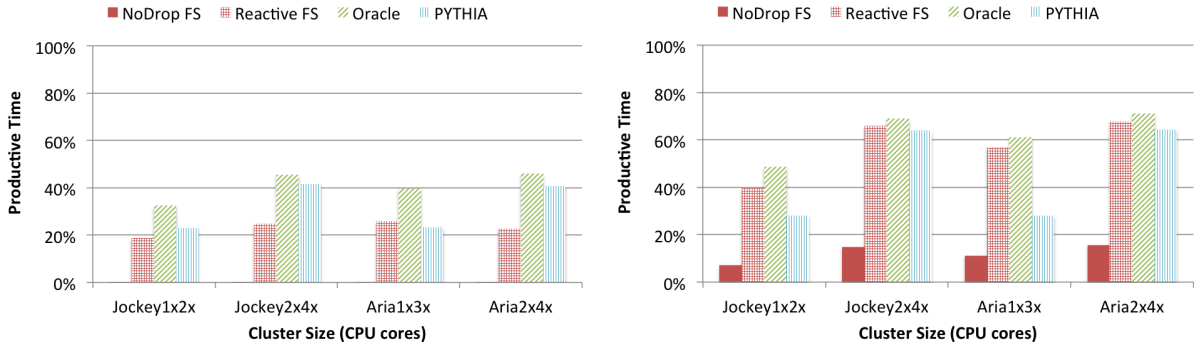
Figure 4.4: Satisfied Deadline Ratio (SDR) with Variable Deadlines for TR1 for cloud capacities of 2250 CPUs (left graph) and 4500 CPUs (right graph). Experiments denoted as 'Jockey' have deadline multiples picked randomly from a set with two values $(1, 2)$ and $(2, 4)$. Experiments denoted as 'Aria' have deadline multiples drawn from uniformly distributed intervals $[1, 3]$ and $[2, 4]$.

fair share is still sufficient to meet deadlines. However, these jobs contribute minimally to PTR. As a result, the productive time achieved from the fair share allocators degrades rapidly as cloud capacity decreases even while meeting some deadlines.

The results from the PYTHIA allocator indicate that even a simple allocation and admission control strategy can restore the number of satisfied deadlines and useful work to near optimal. For unconstrained scenarios, all allocators perform similarly for both deadlines satisfied and productive computation time. Thus, PYTHIA can be used in either scenario to achieve the greatest deadline satisfaction.

4.4.2 Variable Deadlines

Next we consider the variable deadline assignments detailed under **Deadline Types** in Section 4.2. Due to space constraints, we present results only for constrained cloud settings (2250 and 4500 cores) and for TR1. Our results for TR2 are similar to TR1 results and our results for unconstrained clouds show that all allocators behave similarly



(a) Cloud Capacity: 2250 CPU cores

(b) Cloud Capacity: 4500 CPU cores

Figure 4.5: Productivity Time Ratio (PTR) with Variable Deadlines for TR1 for cloud capacities of 2250 CPUs (left graph) and 4500 CPUs (right graph). Experiments denoted as 'Jockey' have deadline multiples picked randomly from a set with two values $(1, 2)$ and $(2, 4)$. Experiments denoted as 'Aria' have deadline multiples drawn from uniformly distributed intervals $[1, 3]$ and $[2, 4]$.

to each other.

Figure 4.4 shows the SDR for each allocator when we employ variable deadlines for two resource-constrained clouds. Using this deadline assignment, PYTHIA satisfies a similar number of deadlines as the Oracle allocator in both cases. In the worst case, PYTHIA is within 5% of the oracle. PYTHIA satisfies up to 177% more deadlines than Reactive FS on the 2250 CPU cloud and up to 58% more deadlines than Reactive FS on the 4500 CPU cloud.

Figure 4.5 presents the PTR for these scenarios. Although PYTHIA meets significantly more deadlines than Reactive FS, it outperforms its PTR only for a subset of the deadline classes. This difference results because PYTHIA must be conservative (i.e. over-provisioning or dropping more jobs) to ensure that they meet their deadlines. That is, PYTHIA computes the `minReqRate` for jobs that now have deadlines that range from very strict (e.g. multiple of 1 which means the deadline is the same as its optimal runtime) to very loose (3 times this value). To ensure that as many jobs as possible meet their dead-

line and there is no wasted work, Pythia conservatively chooses a CPU allocation that will likely enable jobs with strict deadlines to succeed. For jobs with loose deadlines, this results in over-provisioning, preventing other jobs from sharing the cloud. In contrast, the *Oracle* knows the `minReqRate` that each job requires to satisfy its deadline and thus enables more free capacity on the cluster that can be used by other jobs. For cloud capacities of 2250 and 4500 CPUs, Reactive FS introduces from 24% to 30% and from 12% to 30% wasted work (computation time spent for jobs that miss their deadlines and are terminated midstream) respectively for the different deadlines of the evaluation. NoDrop FS, wastes 99% of the computation time on the cloud with 2250 CPUs and from 84% to 93% on the cloud with 4500 CPUs for the different deadline types because it doesn't drop jobs even after their deadline has passed. In contrast, Pythia keeps the wasted work near 0% for all the different deadlines and cluster capacities, because it adapts every time it encounters a stricter deadline. As part of future work, we are investigating ways to adapt Pythia's `CPUFrac` over time to improve its PTR performance.

4.5 Related Work

On this section we present research done on performance prediction for Hadoop and Spark and discuss the key differences to our approach. We then present current approaches for resource allocation under mutli-analytics settings and discuss how Pythia differentiates to support deadline driven workloads. Lastly, we discuss and compare Pythia to previous research efforts that also utilize admission control.

Performance prediction: In order to allocate the required resources and meet job deadlines, much related work focuses on exploiting historic [17, 13, 83, 18, 84, 92, 19], and runtime [14, 80, 17, 13, 14, 81, 82, 18, 92, 19] job information, while other research [94, 82, 85, 20, 78, 19] focuses on building job performance profiles and scalability models

offline. Although, effective in many situations, we show that approaches similar to these suffer when used under resource constrained settings.

Strategies that depend solely on repeated jobs, by definition, do not guarantee performance of ad-hoc queries. While approaches that use runtime models, sampling, simulations, and extensive monitoring, impose overheads and additional costs. Moreover, trace analysis in this chapter and other research [20] shows that some production clouds have small ratio of repeated jobs and these jobs have often large execution times dispersion. Therefore, approaches based on past executions might not have the required mass of similar jobs over a short period of time in order to predict with high statistical confidence. Furthermore, the vast number of jobs have very short computation times [20, 45, 47, 74, 95]. Thus, approaches that adapt their initial allocation after a job has already started might be ineffective. Lastly, most of these approaches require task-level information, for the specific framework they target, either Hadoop [80, 94, 17, 13, 14, 81, 82, 83, 18, 84, 92, 85] or Spark [15, 16]. For this reason, they cannot be used on top of resource managers like Mesos.

Pythia in contrast, does not depend on job repetitions and can therefore target clouds with more diverse workloads; it does not impose overheads to perform extensive runtime monitoring or use job sampling or offline simulations to predict performance. Pythia is also framework-independent because it does not require modeling of the different stages of any particular big data framework.

Sharing on Multi-tenant resource allocators: Cluster managers like Mesos [48] and YARN [9] enable the sharing of cloud and cluster resources by multiple data processing frameworks. Recent research [32, 33, 34] builds on this sharing, to allow users to run jobs without knowledge of the underlying data processing engine. In these multi-analytics settings, the goal of the resource allocator is to provide performance isolation to frameworks by sharing the resources between them [96, 11, 87, 12]. The sharing policies

in these works are deadline-agnostic. To meet deadlines, administrators currently use dedicated clouds, statically split their clouds with the use of a capacity scheduler [96], or require users to reserve resources in advance [97, 98]. Such approaches incur costs for additional clouds or are inefficient and impractical, as they limit peak cloud capacity. Moreover, resources are underutilized when critical jobs are not running.

Another issue encountered in multi-analytics systems, is that frameworks like Hadoop and Spark that run on top of these resource allocators have their own intra-job schedulers that greedily occupy the resources allocated to them, even when they are not using them [25, 79, 67]. *CARBYNE* [79] attempts to address this issue by exploiting task-level resource requirements information and DAG dependencies. It also uses prior runs of recurring jobs to estimate task demands and durations. Then, it intervenes both at the higher level, on the resource allocator, and internally, on framework task schedulers. It withholds a fraction of job resources from jobs that do not use them while maintaining similar completion times. Pythia in contrast, introduces framework-independent admission control that these resource allocators can use to support dynamic sharing of the cloud and deadline driven workflows for either Hadoop or Spark jobs, without requiring task-level information or depending on recurring jobs.

Admission control: Admission control has been suggested as a cloud solution for SaaS providers to effectively utilize their clusters and meet Service Level Agreements (SLAs) [99, 100], to provide map-reduce-as-a-service [101], and to resolve blocking caused by greedy YARN allocations [67]. Pythia is similar in that but targets multi-analytics, resource-constrained clouds. We design Pythia for use by resource managers for deadline-driven big data workloads, to be framework and task independent.

4.6 Summary

In this chapter we present Pythia, a novel admission control system for resource negotiators for big data multi-analytics systems, such as Apache Mesos and YARN. Pythia adds support for deadlines and facilitates more effective resource use when resources are constrained due to physical limitations (private clouds) and cost contraints (when using public clouds), two increasingly common big data deployment scenarios. Pythia is a black box, framework agnostic approach that estimates the CPU resources that a job requires to meet its deadline, based on historic deadline and runtime information of completed jobs. Pythia admits jobs when the estimated resources are available and drops them when their deadlines are no longer feasible. Thus, any job not admitted "fails fast" and wastes no resources, enabling more jobs to be admitted and meet their deadlines.

We empirically evaluate Pythia for different cloud capacities and deadline assignments, using trace-based simulation, driven by production YARN traces. Our results show that Pythia is able to meet significantly more deadlines compared to existing fair-share approaches. Unlike them wasting a big portion of the computation time for jobs that don't meet their deadlines, Pythia almosts eliminates resource waste while maintaining high levels of completed useful work, for the workloads, cloud capacities, and deadline assignments that we consider. Moreover, we have implemented a prototype of Pythia as an allocation module [93] for Apache Mesos 0.27.2 which we will make available following publication.

As part of future work, we are exploring how to extend Pythia to adapt to deadline variability and changing resource constraints. Pythia currently is conservative and only increases its resource provisioning rate (`CPUFrac`) as it encounters tighter deadlines. This attempts to minimize wasted computation, but also leaves room for improvement (e.g. in the PTR (useful work) metric). We are currently considering how to decrease

this rate when we discover that it has been over-conservative using a sliding window and by differentiating between long and short running jobs. We are also considering an approach that incentivizes users to assign realistic deadlines to jobs, both to avoid over allocating resources and to protect from unnecessary increases on the resource provision rate. Finally, we are extending Pythia to consider overheads that lead to imperfect parallelization in our calculation of the minimum resource requirements of jobs, by tracking and incorporating divergence in observed and estimated job runtimes.chapter

Chapter 5

Justice: A Deadline-aware, Fair-share Resource Allocator for Implementing Multi-analytics

In the previous two chapters, we saw that existing resource allocators cannot preserve fairness and meet deadlines in resource-constrained multi-analytics clusters and we suggested a resource allocator that satisfies deadlines in such settings. Meeting deadlines without wasting valuable cluster resources is important for the deadline-driven IoT workloads that are analyzed in resource-constrained clusters. However, we have not yet examined whether our deadline-driven approach can also support fairness in such settings.

In this chapter, we investigate the efficacy of fair-share schedulers in cluster settings that are used increasingly by applications for the emerging IoT domain. We compare existing fair-share algorithms employed by Mesos [48] and YARN [9] to a new approach, called *Justice*, that uses deadline information for each job and historical workload analysis to improve deadline performance and fairness. Rather than using fairness as the allocation criterion as is done for Mesos and YARN, *Justice* estimates the fraction of

the requested resources that are necessary to complete each job just before its deadline expires. It makes this estimate when the job is submitted using the requested number of resources as the number necessary to complete the job as soon as possible. It then relaxes this number according to a running tabulation of an expansion factor that is computed from an on-line post-mortem analysis of all previous jobs run on the cluster. Because the expansion factor is computed across jobs (i.e. globally for the cluster) each analytics framework receives a "fair penalty" for its jobs, which results in a better fair-share, subject to the deadlines associated with each job. Further, *Justice* "risks" running some jobs with greater or fewer resources than it computes they need so that it can adapt its allocations automatically to changing workload characteristics.

We describe the *Justice* approach and compare it to the baseline allocator employed by Mesos and YARN, to simple intuitive extensions to this allocator, and to a job workload "oracle", which knows precisely (i.e. without estimation error) the minimum number of resources needed for each job to meet its deadline. For evaluation, our work uses large production workload traces from an industry partner that provides commercial big-data services using YARN [1]. The original jobs in this trace were not resource constrained nor did they require completion according to individual deadlines. For these reasons we use a discrete-event, trace-driven simulation to represent how these workloads would execute with significantly fewer cluster resources under different deadline formulations found in related work [91, 92, 17, 13, 20].

Our results show that *Justice* performs similarly to the oracle in terms of fairness and deadline satisfaction, and significantly better than the baseline Mesos and YARN allocator. In addition, *Justice* achieves greater productivity and significantly better utilization than its counter-parts when resources are severely constrained. We next describe *Justice* and its implementation. We then overview our empirical methodology (Section 5.2) and

[1]The partner wishes to remain anonymous for reasons of commercial competitiveness.

the characteristics of the workload we evaluate (Section 5.3). We present our results in Section 5.4. In Sections 5.5 and 5.6, we discuss related work and conclude.

5.1 Justice Design

Justice is a fair-share preserving and deadline-aware resource allocator with admission control for resource negotiators that manage batch applications with deadlines, for resource-constrained, shared clusters. *Justice* employs a black-box, framework-agnostic prediction technique (based on measurements of historical job execution times) to estimate the minimum number of CPUs (parallelism) that a job requires to meet its deadline. In a way that outperforms previous fair-share allocators [11, 12], currently used by resource negotiators like Mesos [48] and YARN [9], *Justice* is able to achieve a "fairer" share while satisfying a greater fraction of deadlines when resources are constrained.

Previous work has shown that many fair-share allocators (possibly designed for the "infinite" resources available in a cloud) fail to preserve fairness when resources are limited [25, 79, 67]. This shortfall is due to, in part, to their greedy allocation (a result of their inability to predict future demand) and lack of corrective mechanisms (ex: job preemption or dropping). Instead, *Justice* improves upon these techniques by proactively adapting to future demand and cluster conditions through its resource allocation and admission control mechanisms.

The existing fair-share allocators are also deadline insensitive. They assume that a job submitted by a user has value to that user regardless of how large the turn-around time may be. For *Justice* we assume (especially since cluster resources may be scarce) that each job is submitted with a "maximum runtime" parameter that tells the resource negotiator when the completion of each job is no longer valuable. Henceforth, we term this parameter the "deadline" for the job. The only assumption we make about the

user specified deadline, is that the deadline is feasible, that is an optimal allocation in an empty cluster is sufficient to complete the job before its deadline. Note that current resource negotiators such as Mesos and YARN do not include a provision for specifying job maximum runtime. Instead, many cluster administrators statically split their clusters with the use of a capacity scheduler [96], or require users to reserve resources in advance [97, 98] to create differentiated service classes with respect to turn-around time. However, such approaches are inefficient and impractical in resource-constrained clusters, as they further limit peak cluster capacity. In contrast, *Justice* incorporates deadline information to drive its resource allocation, admission-control, and job dropping decisions.

5.1.1 Resource Allocation

To determine how many CPUs to allocate to a new job, *Justice* uses execution time data logged for previously executed jobs. *Justice* analyzes each completed job and uses this information to estimate the minimum number of CPUs that the job *would have needed* to have finished by its deadline "just-in-time" (represented as the `deadlineCPUs` variable in Algorithm 3). *Justice* assumes that this minimum required capacity utilizes perfect parallelism (speedup per CPU) and that the number of tasks for a job (the division of its input size and the HDFS block size) is the maximum parallelization possible for the job. We refer to this number as the `requestedTasks` for the job. Therefore, the maximum number of CPUs that can be assigned to any job (`maxCPUs`) at any given time is the minimum between the `requestedTasks` and the total cluster capacity (`cluster_capacity`).

To bootstrap the system, *Justice* admits all jobs regardless of deadline; it allocates `requestedTasks` CPUs to the job. For any job for which there are insufficient resources for the allocation, *Justice* allocates the number of CPUs available. When a job completes

Algorithm 3 *Justice* TRACK_JOB Algorithm

```
1: function TRACK_JOB(compTime, requestedTasks, deadline, numCPUsAllocd, success)
2:     deadlineCPUs = compTime/deadline
3:     maxCPUs = min(requestedTasks, cluster_capacity)
4:     minReqRate = deadlineCPUs/maxCPUs
5:     minReqRateList.add(minReqRate)
6:     MinCPUFrac = min(minReqRateList)
7:     MaxCPUFrac = max(minReqRateList)
8:     LastCPUFrac = numCPUsAllocd/maxCPUs
9:     LastSuccess = success
10:    fractionErrorList.append(minReqRate − LastCPUFrac)
11: end function
```

Algorithm 4 Fraction Calculation

```
1: function CALCULATE_ALLOC_FRACTION
2:     if LastSuccess then
3:         CPUFraq = MinCPUFrac
4:     else
5:         CPUFraq = MaxCPUFrac
6:     end if
7:     fraction = (LastCPUFrac + CPUFraq)/2
8:     return fraction
9: end function
```

(either by meeting or exceeding its deadline), *Justice* invokes the pseudocode function `TRACK_JOB` shown in Algorithm 3.

`TRACK_JOB` calculates the minimum number of CPUs required (`deadlineCPUs`) if the job were to complete by its deadline, using its execution profile available from cluster manager logs. Line 2 in the function is derived from the equality:

$$numCPUsAllocd * jobET = deadlineCPUs * deadline$$

On the left is the actual computation time by individual tasks, which we call `compTime` in the algorithm. `numCPUsAllocd` is the number of CPUs that the job used during execution and `jobET` is its execution time without queuing delay. The right side of the equation is the total computation time consumed across tasks if the job had been assigned `deadlineCPUs`, given this execution profile (`compTime`). `deadline` is the time

Algorithm 5 Fraction Correction and Validation

```
1: function CORRECT_ALLOC_FRACTION(fraction)
2:     correction =CALC_SMOOTHED_AVG(fractionErrorList))
3:     correctedFraction = fraction + correction
4:     correctedFraction =VALIDATE_FRACTION(correctedFraction)
5:     return correctedFraction
6: end function
7: function VALIDATE_FRACTION(fraction)
8:     if fraction < min(minRequiredAllocationRatioList) then
9:         fraction = min(minRequiredAllocationRatioList)
10:    else if fraction > 1 then
11:        fraction = 1
12:    end if
13:    return fraction
14: end function
```

(in seconds) specified in the job submission. By dividing `compTime` by `deadline`, we extract `deadlineCPUs` for this job.

Next, *Justice* divides `deadlineCPUs` by the maximum number of CPUs allocated to the job. The resulting `minReqRate` is a fraction of the maximum that *Justice* could have assigned to the job and still have it meet its deadline. *Justice* adds `minReqRate` to a list of fractions (`minReqRateList`) that contains the minimum required rates (fractions of `deadlineCPUs` over `requestedTasks`) across all completed jobs. Then it calculates from this list the global minimum (`MinCPUFrac`) and maximum (`MaxCPUFrac`) fractions. It also tracks the observed fraction allocated to the last completed job (`LastCPUFrac`) and whether the job satisfied or exceeded its deadline (`LastSucess`). *Justice* then uses `MaxCPUFrac` and `MinCPUFrac` to predict the allocatable fractions of future jobs. `MaxCPUFrac` and `MinCPUFrac` are always less than or equal to 1. The tighter the deadlines, the more conservative (nearer to 1) these fractions and the corresponding *Justice*'s resource provisioning will be.

Then, to compute the cpu allocation fraction (`alocCPUFrac`) for each newly submitted job, *Justice* takes the average of the `LastCPUFrac` and either `MinCPUFrac` or `MaxCPUFrac` as shown in Algorithm 4, depending on whether the last completed job met or violated its

Algorithm 6 Admission Control and Resource Allocation

```
1: function ADMISSION_CONTROL(RequesterJob)
2:     for all j ∈ SubmittedJobs do
3:         Feasible = True, TTD = Deadline − ElapsedTime
4:         reqCpus = ESTIMATE_REQ(j, TTD)
5:         if reqCpus > min(taskCount, capacity) then
6:             Feasible = False
7:         end if
8:         if Share(j) < reqCpus then
9:             if Feasible == True then
10:                priority = reqCpus/TTD, ADD_TO_HEAP(priority, j)
11:            else
12:                DROP_JOB(j)
13:            end if
14:        end if
15:    end for
16:    allocations = ALLOC_RESOURCES(heap)
17:    if RequesterJob ∉ allocations then
18:        Add RequesterJob to queue
19:    end if
20: end function

21: function ESTIMATE_REQ(Job)
22:    maxCpus = min(tasks, capacity), reqCpus = maxCpus
23:    if CompletedJobs > 1 then
24:        fraction = CALCULATE_ALLOC_FRACTION()
25:        fraction = CORRECT_ALLOC_FRACTION(fraction)
26:        fraction = (deadline/(deadline − queue)) ∗ fraction
27:        reqCpus = max(ceil(fraction ∗ maxCpus), 1)
28:    end if
29:    return reqCpus
30: end function

31: function ALLOC_RESOURCES(heap)
32:    offers = CREATE_OFFERS(heap)
33:    allocations = SEND_OFFERS(offers)
34:    return allocations
35: end function

36: function CREATE_OFFERS(heap)
37:    while availableCpus > 0 and heap not empty do
38:        for all Job j ∈ heap do
39:            offer = min(request(j), availableCpus)
40:            if offer < request(j) then
41:                offer = 0
42:            else
43:                availableCpus− = offer
44:                offersDict[j] = offer
45:            end if
46:        end for
47:    end while
48:    return offersDict
49: end function
```

deadline respectively. In other words, consecutive successes make *Justice* more aggressive and it allocates smaller resource fractions (`alocCPUFrac` converges to `MinCPUFrac`) while deadline violations make *Justice* more conservative and it increases the allocated fraction to prevent future violations (`alocCPUFrac` converges to `MaxCPUFrac`).

Justice also utilizes a Kalman filter mechanism to correct inaccuracies of its initial estimations (Algorithm 5). Every time a job completes its execution, *Justice* tracks the estimation error; the divergence of the given allocation fraction from the ideal minimum fraction (`deadlineCPUs`). To correct the allocation estimations, *Justice* calculates a weighted average of the historical errors. It can be configured to assign the same weights to all past errors or use exponential smoothing and "trust" more recent errors. Lastly, a validation mechanism ensures that the corrected fraction is still between allowable limits (the fraction should not be less than the minimum observed `MinCPUFrac` or greater than 1).

After `alocCPUFrac` is calculated, corrected, and validated as described above, *Justice* multiplies `alocCPUFrac` by the number of tasks requested in the job submission (rounding to the next largest integer value). It uses this value (or the maximum cluster capacity, whichever is smaller) as the number of CPUs to assign to the job for execution (Function `estimate_req` in Algorithm 6). If this number of CPUs is not available, it enqueues the job. *Justice* performs this process each time a job is submitted or completes. It also updates the deadlines for jobs in the queue (reducing each by the time that has passed since submission), recomputes the CPU allocation of each enqueued job and drops any enqueued jobs with infeasible deadlines.

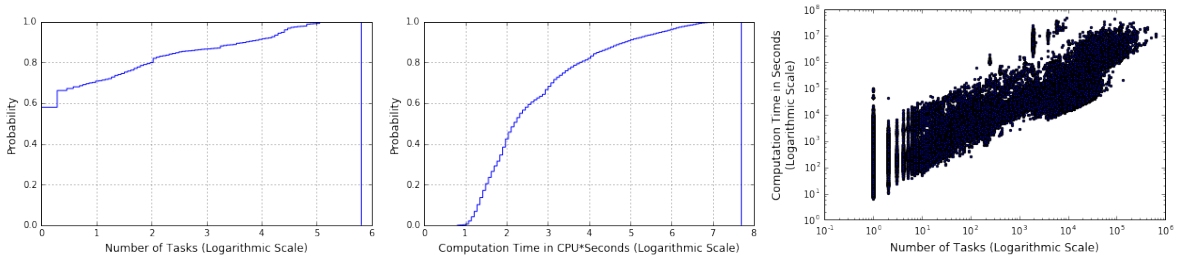
5.1.2 Admission Control

After estimating the resources that jobs need to meet their deadlines, *Justice* implements a *proactive* admission control so that it can prevent infeasible jobs (jobs likely to miss their deadlines) from ever entering the system and consuming resources wastefully. This way, *Justice* attempts to maximize the number of jobs that meet their deadline even under severe resource constraints (i.e. limited cluster capacity or high utilization). *Justice* also tracks jobs that violate their deadlines and selectively drops some of them to avoid further waste of resources. It is selective in that it terminates jobs when their `requestedTasks` exceed a configurable threshold. Thus, it still able to collect statistics on "misses" to improve its estimations by letting the smaller violating jobs complete their execution while at the same time it prevents the bigger violators (which are expected to run longer) from wasting cluster resources.

Justice admits jobs based on a pluggable priority policy. We have considered various policies for *Justice* and use a policy that prioritizes minimizing the number of jobs that miss their deadlines. For this policy, *Justice* gives priority to jobs with a small number of tasks and greatest time-to-deadline. However, all of the policies that we considered (including shortest time-to-deadline) perform similarly. Once *Justice* has selected a job for submission, it allocates the CPUs to the job and admits it to the system for execution. Once a job run commences, its CPU allocation does not change.

5.2 Experimental Methodology

We compare *Justice* to the fair-share allocator that currently ships with the open-source Mesos [48] and YARN [9] using trace-based simulation. Our system is based on Simpy [86] and replicates the execution behavior of industry-provided production traces of big data workloads (cf Section 5.3).



(a) CDFs of number of tasks per job (b) CDFs of computation time per job (c) Job computation time vs number of tasks

Figure 5.1: **Workload Characteristics:** Number of tasks per job, computation time per job, and computation time relative to jobs size (in number of tasks). Small jobs are large in number but consume a very small proportion of trace computation time.

The current Mesos and YARN fair-share allocator does not take into account the notion of deadline. When making allocation decisions, it (tacitly) assumes that each job will use the resources allocated to it indefinitely and that there is no limit on the turn-around time a job's owner is willing to tolerate. We hypothesize a straight-forward modification to the basic allocator that allows it to consider job deadlines (which would need to be submitted with each job) when making decisions.

Finally, we implement an "oracle" allocator that has perfect foreknowledge of the minimum resource requirements each job needs to meet its deadline exactly. Note that the oracle does not implement system-wide prescience – its prediction is perfect on a per-job basis. That is, the oracle does not try all possible combinations of job schedules to determine the optimal allocation. Instead, the oracle makes its decision based on a perfect prediction of each job's needs. These allocation policies are summarized as follows:

`Baseline FS`: This allocator employs a fair sharing policy [12, 87, 88, 89, 90] without admission control. Its behavior is similar to that of the default allocator in Mesos and YARN and, as such, runs all jobs submitted regardless of their deadlines and resource requirements.

`Reactive FS`: This allocator extends Baseline FS by allowing the allocator to terminate any job that has exceeded its deadline. That is, it "reacts" to a deadline miss by freeing the resources so that other jobs may use them.

`Oracle`: This allocator allocates the minimum number of resources that a job requires to meet its deadline. If sufficient resources are unavailable, the Oracle queues the job until the resources become available or until its deadline has passed (or is no longer achievable). For the queued jobs, Oracle gives priority to jobs with fewer required resources and longer time until the deadline.

`Justice`: As described in Section 5.1, this allocator proactively drops, enqueues, or admits jobs submitted. It estimates the share of each job as a fraction of its maximum demand. This fraction is based on the historical performance of jobs in the cluster. For the queued jobs, *Justice* gives priority to jobs with fewer required resources and longer computation times. *Justice* drops any jobs that are infeasible based on a comparison of their deadlines with a prediction of the time to completion. Jobs that are predicted to miss their deadlines are not admitted (they are dropped immediately) as are any jobs that exceed their deadlines.

5.2.1 Deadline Types

We evaluate the robustness of our approach by running experiments using deadline formulations from prior works [91, 92, 17, 13, 20] and interesting variations on them. In particular, we assign deadlines that are multiples of the optimal execution time of a job (which we extract from our workload trace). We use two types of multiples: Fixed and variable.

`Fixed Deadlines:` With fixed deadlines, we use a deadline that is a multiple of the optimal execution time (a formulation found in [91, 92]). Each deadline is expressed as

$D_i = x \cdot T_i$, where T_i is the optimal runtime of the job and $x >= 1.0$ is some fixed multiplicative expansion factor. In our experiments, we use constant factors of $x = 1$ and $x = 2$, which we refer to as *Fixed1x* and *Fixed2x* respectively.

`Variable Deadlines:` For variable deadlines, we compute deadline multiples by sampling distributions. We consider the following variable deadline types:

- *Jockey*: We pick with equal probability a deadline expansion factor x from two possible values (a formulation described in [20]). In this work, we use the intervals from the sets with values $(1, 2)$ and $(2, 4)$ to choose x and, again, compute $D_i = x \cdot T_i$, where T_i is the minimum possible execution time. We refer to this variable deadline formulation as *Jockey1x2x* and *Jockey2x4x*.

- *90loose*: This is a variation of the Jockey1x2x deadlines, in which the deadlines take on the larger value (i.e. are loose) with a higher probability (0.9) while the other uses the smaller value.

- *Aria*: The deadline multiples of this type are uniformly distributed in the intervals $[1, 3]$ and $[2, 4]$ (as described in [17, 13]); we refer to these deadlines as *Aria1x3x* and *Aria2x4x*, respectively.

5.3 Workload Characterization

To evaluate *Justice*, we use a 3-month trace from production Hadoop deployments executing over different YARN clusters. The trace was recently donated to the *Justice* effort by an industry partner on condition of anonymity. The trace contains a job ID, job category, number of map and reduce tasks, map and reduce time (computation time

CPUs	**Jobs**	**Comp. Time (Hours)**	**1-Task Pct**	**1-Task Time Pct**
9345	159194	8585673	58%	0.1%

Table 5.1: Trace Summary. Columns are peak cluster capacity, total number of jobs, total computation time in hours, percentage of 1-task jobs, and percentage of 1-task job computation time.

across tasks), job runtime, among other data. It does not contain information about the scheduling policy or HDFS configuration used in each cluster. Thus we assume a minimum of one CPU per task and use this minimum to derive cluster capacity; we are considering sub-portions of CPUs (vcores) as part of future work. *Justice* uses the number of map tasks (as `requestedTasks` in Algorithm 3) and map time (as `compTime` in Algorithm 3).

Table 5.1 summarizes the job characteristics of the trace. The table shows the peak cluster capacity (total number of CPUs), the total number of jobs, the total computation time across all tasks in the jobs, the percentage of jobs that have only one task, and the percentage of computation time that single-task jobs consume across jobs. There are 159194 jobs submitted and the peak observed capacity (maximum number of CPUs in use) is 9345[2].

The table also shows that even though there are many single-task jobs, they consume a small percentage of the total computation time. To understand this characteristic better, we present the cumulative distribution of number of tasks in Fig. 5.1a and computation

[2]We have tested *Justice* on a second trace that contains more than 1 million job entries from the same industry partner. The distribution properties of job sizes are remarkably similar to the trace we have chosen to use. However, because many of the jobs in this larger trace repeat (creating more autocorrelation in the job series), we believe that the smaller trace is a greater challenge for *Justice*'s predictive mechanisms. The results for this larger trace are, indeed, better than the results we present in this chapter. We have omitted them for brevity.

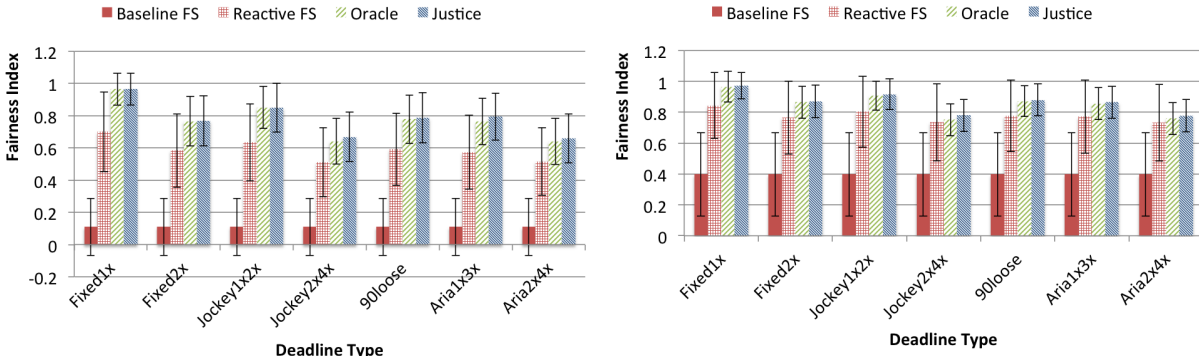
time in Fig. 5.1b per job in logarithmic scale. Approximately 60% of the jobs have a single task and 70% of the jobs have fewer than 10 tasks. Only 13% of the jobs have more than 1000 tasks. Also, the vast majority of jobs have short computation times. Approximately 70% of jobs have computation time that is less than 1000 CPU*seconds, i.e. their execution would be 1000 seconds if they were running in one CPU core.

The right graph in the figure compares job computation time with the number of tasks per job (both axes are on a logarithmic scale). 80% of the 1-task jobs and 60% of the 2-10 task jobs have computation time of fewer than 100 seconds. Their aggregate computation time is less than 1% of the total computation time of the trace. Jobs with more than 1000 tasks account for 98% of the total computation time. Finally, job computation time varies significantly across jobs.

We have considered leveraging the job ID and number of map and reduce tasks to track repeated jobs, but find that for this real-world trace such jobs are small in number. 18% of the jobs repeat more than once and 12% of the jobs repeat more than 30 times. Moreover, we observe high performance variation within each job class. Previous research has reported similar findings and limited benefits from exploiting job repeats for production traces [20].

5.4 Results

We evaluate *Justice* using the production trace for different resource-constrained cluster capacities (number of CPUs). We compare *Justice* against different fair share schedulers and an Oracle using multiple deadline strategies: a fixed multiple (Fixed), a random multiple (Jockey), a uniform multiple (Aria) of the actual computation time, and mixed loose and strict deadlines (90loose), as described on Section 5.2.



(a) Fairness Index with 2250 CPUs

(b) Fairness Index with 4500 CPUs

Figure 5.2: **Fairness Evaluation:** Average of Jain's fairness index applied to the fraction of demand (and 0.95 error bars) with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph). Experiments denoted as 'Fixed' have deadlines multiples of 1 and 2. Experiments denoted as 'Jockey' have deadline multiples picked randomly from a set with two values (1, 2) and (2, 4). Experiments denoted as '90loose' have 90% deadlines with a multiple of 2 and 10% deadlines with a multiple of 1. Experiments denoted as 'Aria' have deadline multiples drawn from uniformly distributed intervals [1, 3] and [2, 4]

5.4.1 Fairness Evaluation

We use Jain's fairness index [102] applied to the fraction of demand each scheduler is able to achieve as a measure of fairness. For each job i, among n total jobs, we define the fraction of demand as $F_i = \frac{A_i}{D_i}$ where D_i is the resource request for job i and A_i is the allocation given to job i. When $A_i >= D_i$ the fraction is defined to be 1. Jain's fairness index is then $\frac{|\sum_{i=1}^{n} F_i|^2}{n * \sum_{i=1}^{n} F_i^2}$.

Figure 5.2 presents the fairness index averaged over 60-sec intervals for all the allocation policies and deadlines considered in this study and for two resource-constrained cluster sizes; very constrained cluster with 2250 CPUs (left graph) and moderately constrained cluster with 4500 CPUs (right graph). The results show that in resource constrained settings, fair-shair allocation policies generate substantially lower fairness indices compared to *Justice*.

In resource constrained clusters, when CPU demands exceed available cluster resources, fair-share mechanisms can violate fairness. This occurs because these mechanisms do not anticipate the arrival of future workload. Thus jobs that require large fractions of the total resource pool get their full allocations, causing jobs that arrive later to block or to be under-served [25]. Moreover, jobs that are waiting in queue may miss their deadlines while waiting (i.e. receive an A_i value of zero) or receive an under allocation once they are released.

Note that adding the ability to simply drop jobs that have missed their deadlines does not alleviate the fairness problem entirely. The Reactive FS policy (described in Section 5.2) achieves better fairness than the Baseline fair-share scheduler, but does not achieve the same levels as *Justice*. When a large job (one with a large value of D_i) can meet its deadline (i.e. it is not dropped by Reactive FS), it may only get a small fraction of its requested allocation (receiving a small value of A_i) thereby contributing to the fairness imbalance when compared to *Justice*. Because the confidence intervals between Reactive FS and *Justice* overlap, we also conducted a Welch's t-test [103] for all deadline-types and cluster sizes. We find that in all cases, the P-value is very small (e.g. significantly smaller than 0.01). Thus the probability that the means are the same is very low.

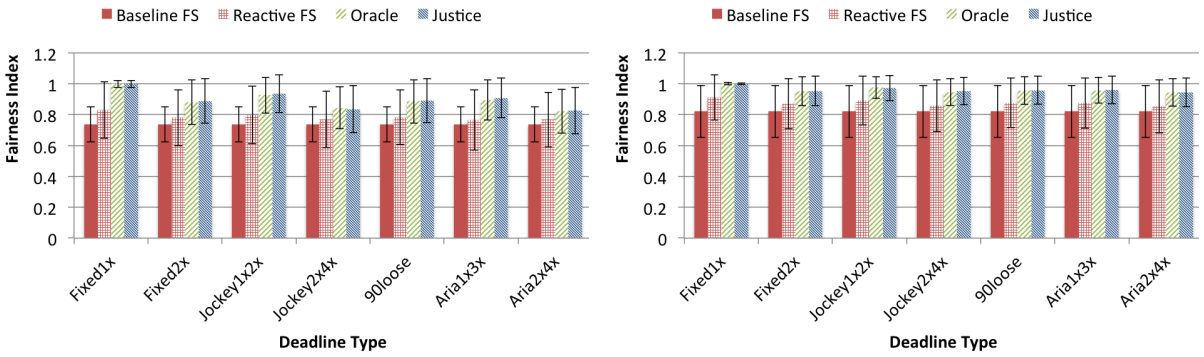
The reason *Justice* is able to better maintain fairness is that it predicts the possibility of future demand and uses these predictions to implement admission control. *Justice* uses a running tabulation of the average fraction of A_i/D_i that was required by previous jobs to meet their deadline, to weight the value of A_i/D_i for each newly arriving job. *Justice* computes this fraction globally by performing an on-line "post mortem" of completed jobs. Then, for each new job, *Justice* allocates a fraction of the demand requested using this estimated fraction. *Justice* constantly updates its estimate of this fraction so that it can adapt to changing workload conditions. As a result, every requesting job gets the

same share of resources as a fraction of its total demand, which is by definition the best possible fairness according to Jain's formula used above.

For this reason, the fairness achieved by *Justice* is better for variable deadlines (e.g., Aria1x3x) even compared to the Oracle. The Oracle allocates to every job the minimum amount of resources required to meet the deadline. Consequently, when the deadline tightness across jobs differ, the fraction of resources that each job gets compared to its maximum resources will also differ. This leads to inequalities in terms of fairness. To avoid the paradox of an Oracle not giving perfect fairness, we could modify Jain's formula by replacing the maximum demand of a job with the minimum required resources in order to meet a deadline. However, we wish to use prior art when making comparisons to the existing fair-share allocators, and so the Oracle (under this previous definition) also does not achieve perfect fairness. In other words, Oracle is an oracle with respect to minimum resource requirements needed to satisfy each job's deadline and not a fairness oracle for the overall system.

Although *Justice* yields the best fairness results compared to other allocators, it is not optimal (i.e. the fairness index is not 1). In particular, when queued jobs are released they may miss their deadlines, but while doing so, cause other jobs to receive little or no allocation. To compensate for this, *Justice* attempts to further weight their allocation by the ratio of the deadline to the time remaining to the deadline ($\frac{deadline}{deadline-queueTime}$), or if achieving the deadline is not possible, *Justice* drops them to avoid wasted occupancy. The cost of this optimization is an occasional fairness imbalance but this cost is less than that for the other allocators we evaluate.

Integer CPU assignment is another source of fairness imbalance. Because jobs require an integer number of CPUs each allocation must be rounded up when it is weighted by the current success fraction. For small jobs, the additional fraction constitutes a significant overhead in terms of fairness. While the industry traces contain large numbers of small



(a) Fairness Index with 2250 CPUs

(b) Fairness Index with 4500 CPUs

Figure 5.3: **Equality Evaluation:** Average equality indexes using Jain's fairness index with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph).

jobs, they are often short lived allowing *Justice* to adapt overall fairness quickly. We are considering sub-CPU allocations as part of future work.

"Equality" does not mean "Justice"

In this subsection we discuss what form of "fairness" the fair-share allocators are preserving by the way they operate and compare against *Justice*. Fair-share allocators are trying to give an "equal" share across jobs running on the cluster and they completely ignore the number of resources they require to meet their deadlines. Herein, we will refer to this form of fairness as "equality". Equality, however, should not be the desired property in deadline-driven workloads because it treats all jobs as "same" (with same maximum demand), but in reality these workloads have jobs with different priorities and diverse deadline tightness.

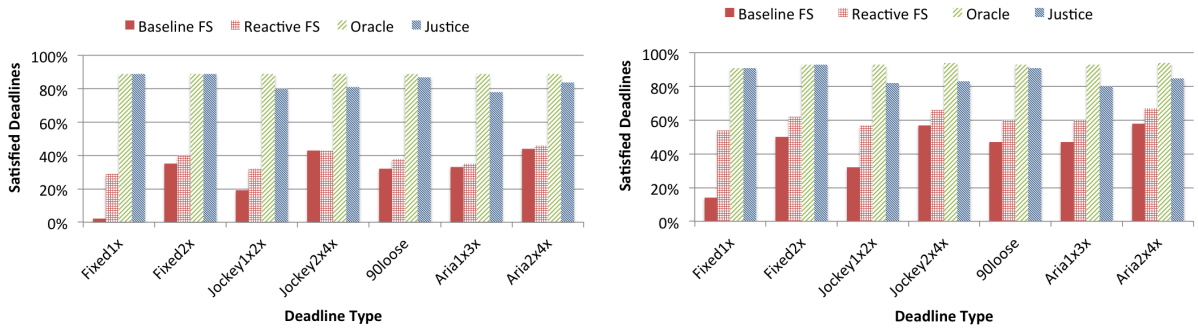
To evaluate equality, we use again Jain's fairness index [102] as defined earlier on this section, i.e., $\frac{|\sum_{i=1}^{n} F_i|^2}{n*\sum_{i=1}^{n} F_i^2}$, but this time F_i corresponds to the resource allocation of each job in the cluster and not to the fraction of demand. We classify jobs in classes

based on their maximum demand, calculate the index for each job, and then calculate the weighted average of these indexes, with weights corresponding to the number of jobs on each demand class (e.g., all jobs with demand of x CPUs). We do this classification to avoid considering as "unfair" (or "unequal"), allocations with job share inequalities existing not due to the allocation mechanism but due to the corresponding differences in the maximum demand (A job cannot get more than what it demands to achieve the same share with a higher-demand job).

Figure 5.3 presents the equality results across all allocators and cluster sizes we consider in this chapter. The results show that in resource constrained settings, fair-shair allocation policies are preserving equality much better than they do for fairness, but they are still doing worse compared to *Justice*. *Justice* outperforms the fair-share allocators by up to 23% on the 2250 cluster and 17% on the 4500 cluster. Even though the goal of *Justice* is not to preserve equality but instead prioritize for fairness, it still does better compared to the fair-share allocators for two reasons; First, it keeps the cluster less utilized (Discussed later on Section 5.4.4) and therefore less jobs are waiting in queue to contribute negatively to equality (as they don't get any resources at all). Second, due to constrained resources, *Justice* drops more frequently bigger jobs (jobs with higher demands). This, eliminates another factor that contributes negatively to the equality index as the equality differences across bigger jobs can be greater compared to the ones of smaller jobs.

5.4.2 Deadline Satisfaction

We next evaluate how well the allocators perform in terms of deadline satisfaction. Our goal with this set of experiments is to verify that *Justice* is not simply achieving fairness by dropping a large fraction of jobs – so that those that remain receive a fair



(a) Satisfied Deadlines with 2250 CPUs

(b) Satisfied Deadlines with 4500 CPUs

Figure 5.4: **Deadline Satisfaction:** Satisfied Deadlines Ratio (SDR) with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph) for different deadline types.

allocation.

To investigate this, we compute the *Satisfied Deadline Ratio (SDR)* as the fraction of the jobs that complete before their deadline over the total number of submitted jobs. For the set of all the submitted jobs $J_1, J_2, ..., J_n$, if $m < n$ is the subset of successful jobs $J_1, J_2, ...J_m$, then SDR is: $\frac{\sum_{i=1}^{m} J_i}{\sum_{j=1}^{n} J_j}$.

Figure 5.4 presents the SDR for each combination of allocator and deadline type. For all deadline types, *Justice* meets significantly more deadlines than the fair-share policies and performs similarly to the Oracle. *Justice* satisfies at least 88% more deadlines than Baseline FS and from 83% to 207% more deadlines than Reactive FS. *Justice* outperforms fair-share policies because these policies do not consider deadline information and share resources naively and greedily. Because *Justice* is able to use both job deadlines and historical job behavior in its allocation decision, it is able to meet a larger fraction of deadlines than existing allocators while achieving greater fairness.

In particular, without admission control, the Baseline and Reactive FS allocators must admit a large fraction of jobs that ultimately do not meet their deadlines. This "wasted" work has two consequences on deadline performance. First, it causes unnecessary queuing

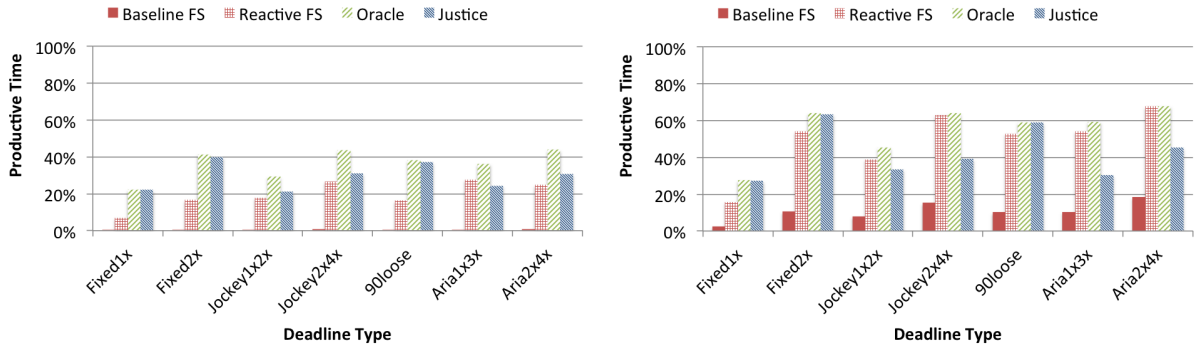
of jobs that, because of the time spent in queue, may also miss their deadlines. Second, it causes resource congestion, thereby reducing the fraction of resources allocated to all jobs. Consequently, some jobs, which would otherwise succeed, miss their deadlines. By attempting to identify those jobs most likely to miss and dropping those jobs proactively, *Justice* is able to achieve a larger fraction of deadline successes overall.

Fair-share policies also fail to meet more deadlines also because of their use of greedy allocation. They allocate as many resources as are available until they run out of resources regardless of what jobs require to meet their deadlines. As a consequence, jobs with looser deadlines get more resources than what they actually need to finish by their deadline, wasting valuable resources that are needed for future jobs with tighter deadlines. In contrast, *Justice* attempts to identify, based on the fraction of demand that previous successful jobs needed in order to meet their deadlines, the *minimum number* of resources required to meet their deadlines "just in time."

Finally, as noted previously, the Oracle does not have perfect information (i.e., it does not have a global optimal schedule). Instead it knows the actual job computation time (`compTime`). Thus, it is able to assign the minimum number of CPUs to each job to satisfy its deadline. SDR for Oracle is not 100% because it must drop (refuse to admit) jobs for which there is insufficient capacity to meet their deadline.

5.4.3 Efficient Resource Usage

We next evaluate workload productivity, i.e. the measure of productive time (i.e. the work done by jobs that complete by their deadlines) and wasted time (i.e. work done by jobs that miss their deadline) via the metrics *Productive Time Ratio (PTR)* and *Wasted Time Ratio (WTR)*. For the set of all the submitted jobs $J_1, J_2, ..., J_n$ and their corresponding runtimes $T_1, T_2, ..., T_n$ we consider the subset of $m < n$ successful jobs



(a) Productive Time with 2250 CPUs

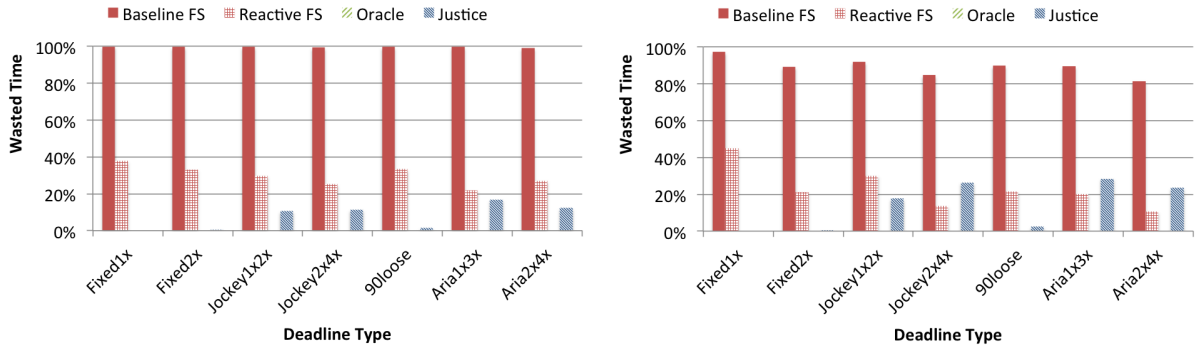
(b) Productive Time with 4500 CPUs

Figure 5.5: **Productivity:** Productive Time Ratio (PTR) with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph) for different deadline types.

$J_1, J_2, ..., J_m$ and the subset of $k < n$ failed or dropped jobs $J_1, J_2, ..., J_k$ where $n = m+k$. PTR is $\frac{\sum_{i=1}^{m} T_i}{\sum_{j=1}^{n} T_j}$ and WTR is $\frac{\sum_{i=1}^{k} T_i}{\sum_{j=1}^{n} T_j}$.

Figure 5.5 and Figure 5.6 present PTR and WTR, respectively, for different allocation policies and deadline type for two constrained clusters. For all cases, Baseline FS spends a very small ratio of computation time productively, i.e. it spends almost all the computation time on jobs that missed their deadlines. Reactive FS improves over Baseline FS by reactively dropping jobs that have already violated their deadlines. *Justice*, performs significantly better (up to 221% higher PTR and up to 100% lower WTR than Reactive FS) and slightly worse than the Oracle (up to 33% lower PTR) for the 2250 CPUs cluster. *Justice* outperforms fair-share policies because it proactively drops jobs with violated deadlines and jobs that it predicts are likely to miss their deadline.

Our experiments also show that the more constrained or utilized the cluster is, the better *Justice* performs in terms of PTR and WTR, relative to the other allocators we consider. Baseline FS fails to satisfy deadlines of bigger jobs because it shares a very limited resources equally between bigger and smaller jobs. This share, under resource



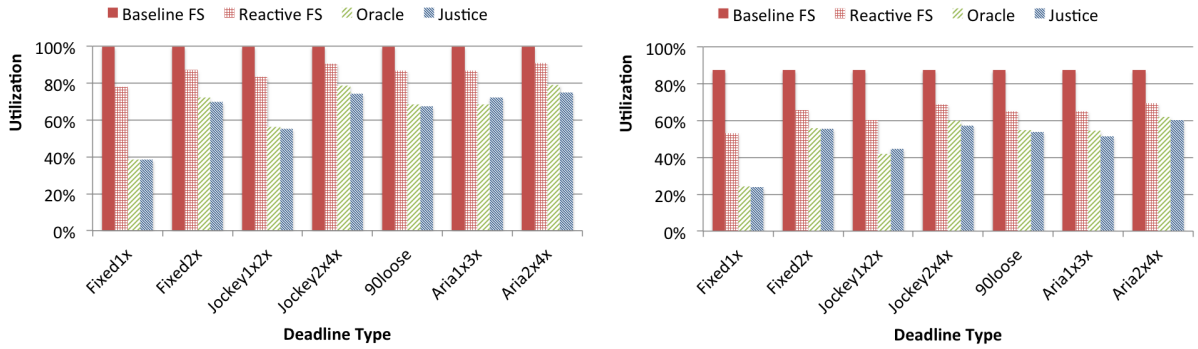
(a) Wasted Time with 2250 CPUs

(b) Wasted Time with 4500 CPUs

Figure 5.6: **Resource Waste:** Wasted Time Ratio (WTR) with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph) for different deadline types. Lower is better.

constrained settings, is not sufficient for the bigger jobs to complete on time. Reactive FS improves PTR and WTR because it drops jobs that violate their deadlines, freeing up resources for other jobs. *Justice* wastes significantly fewer resources compared to Reactive FS because it drops jobs with large expected computation times using its pluggable priority policy (Section 5.1), as soon as they become infeasible.

When the cluster is less constrained (e.g. 4500 CPUs in the right graphs), *Justice*'s PTR is still significantly better than Baseline FS (from 146% on Aria2x4x up to 926% on Fixed1x). It also outperforms Reactive FS up to 72% and performs similarly to the Oracle, for deadline types with less variation (Fixed and 90loose). However, it achieves slightly (14% for Jockey1x2x) or moderately (44% for Aria1x3x) less PTR for high variable deadline types, even though it still satisfies significantly more deadlines compared to Reactive FS for the these deadline types (recall *Justice*'s SDR on Figure 5.4 is 44% and 33% higher than reactive FS for Jockey1x2x and Aria1x3x respectively). Specifically, as resource scarcity is reduced for a fixed workload, large jobs that are admitted by the Baseline FS and Reactive FS allocators stand a better chance of getting the "extra"



(a) Utilization with 2250 CPUs

(b) Utilization with 4500 CPUs

Figure 5.7: **Cluster Utilization:** Utilization with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph) for different deadline types.

resources necessary to complete, and thus, add to the PTR compared to *Justice*, which might have excluded them due to admission control. However, when deadlines are variable, *Justice*'s admission control is conservative, prioritizing fairness and deadline success over resource saturation. This result indicates that extant fair-share allocators may be more appropriate for maximizing productive work when resources are more plentiful and the need to meet deadlines less of a concern. Put another way, when resources are plentiful, the cost of meeting a higher fraction of deadlines with greater fairness is a lower PTR due to admission control.

5.4.4 Cluster Utilization

The final set of experiments that we perform investigate how allocation policies for resource constrained clusters impact cluster utilization and CPU idle times. *Justice* considers a CPU to be *idle* when the allocator has not assigned to it any tasks to run and and to be *busy* when the CPU is running a task. We then define *Cluster Utilization* as $\frac{busy}{idle+busy}$ where *busy* is the total busy time and *idle* is the total idle time across a

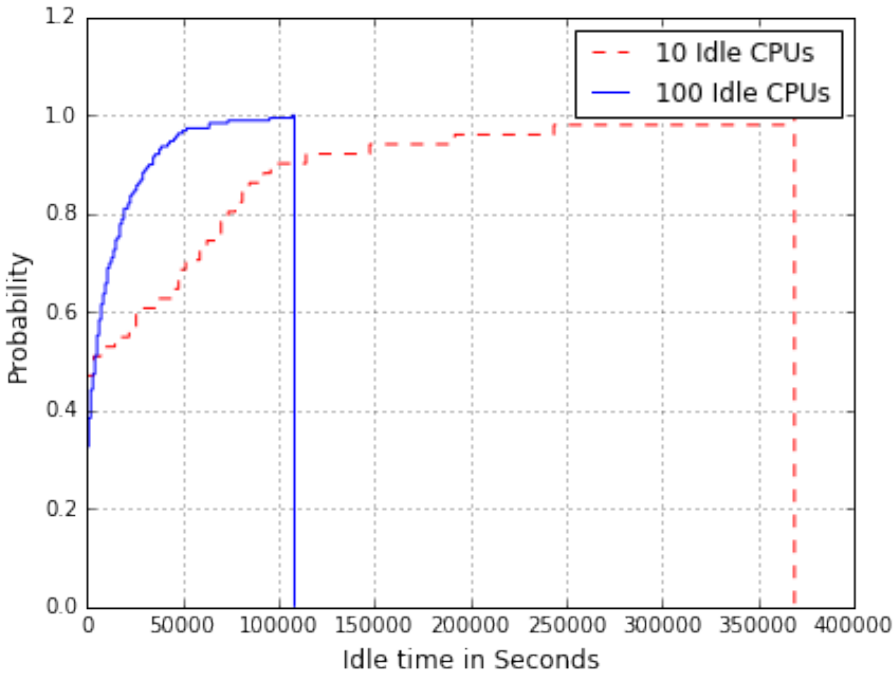


Figure 5.8: CDFs of idles times of 10 and 100 CPU groups.

workload.

Figure 5.7 shows the cluster utilization for 2250 CPU (left graph) and 4500 CPU (right graph) cluster sizes, for the different deadline types that we consider. The results in the left graph (2250 CPU) are particularly surprising and somewhat counter-intuitive. Given severe resource constraints, *Justice* achieves lower utilization than the other allocators, but (as presented previously in Figures 5.5a and 5.6a respectively) exhibits higher PTR and lower WTR. So *Justice* enables more productive work with less waste and less cluster utilization. One might assume that the utilization difference is due to less productivity or more overhead but as these results show, the lower utilization is simply because *Justice* does not need the capacity to meet a greater fraction of deadlines (cf Figure 5.4a) while achieving greater fairness (cf Figure 5.2a).

These results are also interesting in that they reveal a potential opportunity to introduce *more* workload (to take advantage of the available utilization that is not used

by *Justice*) when resources are severely constrained. To investigate this potential, we extract and analyze the number and duration of idle CPUs that correspond to the experiments shown in Figure 5.7a for the Aria1x3x deadline type. Figure 5.8 presents the cumulative distribution of idle time for CPUs that are simultaneously idle in groups of 10 (red, dotted curve) and 100 (blue, solid curve). We find that for deadline types that yield lower utilizations, idle time durations are even larger; we omit these results for brevity.

From these results, we observe that 81% of the 10-CPU groups remain idle more than 100 seconds, 68% more than 500 seconds and 59% more than 1000 seconds. Similarly for 100-CPU groups, 80%, 52%, and 41% have idle times of 100 seconds, 500 seconds and 1000 seconds, respectively. From these results, we can derive that 10-CPU and 100-CPUS idle groups exist at any given time of the trace duration with probabilities 98% and 76% respectively.

We next consider the workload characteristics of the trace that we study (Section 5.3). We have shown (Figure 5.1b) that 40% of jobs compute for less than 100 CPU*seconds, 60% compute for less than 500 CPU*seconds, and 70% compute less than 1000 CPU*seconds. Moreover, approximately 60% of the jobs employ a single task, 70% of the jobs have fewer than 10 tasks, and 80% less than 100 tasks (Figure 5.1a). As a result, *Justice* is able to free up enough cluster capacity for sufficient durations to as to admit significant additional workload. That is, if the trace contained more jobs with these characteristics, *Justice* would likely have been able to achieve similar fairness, deadline, and productive work results via increased utilization. We are currently investigating this potential and how to best exploit it as part of on-going and future work.

5.5 Related Work

On this section we present current approaches for resource allocation under multi-analytics settings and discuss how PYTHIA differentiates to support deadline driven workloads. We then present research done on performance prediction for Hadoop and Spark and discuss the key differences to our approach. Lastly, we discuss and compare *Justice* to previous research efforts that also utilize admission control.

Sharing on Multi-tenant Resource Allocators: Cluster managers like Mesos [48] and YARN [9] enable the sharing of cluster resources by multiple data processing frameworks. Recent research [32, 33, 34] builds on this sharing, to allow users to run jobs without knowledge of the underlying data processing engine. In these multi-analytics settings, the goal of the resource allocator is to provide performance isolation to frameworks by sharing the resources between them [96, 11, 87, 12]. However, under resource-constrained cluster settings, the fair-shair policies [11, 12] fail to preserve fairness (Section 5.4.1). Also, all the sharing policies in these works are deadline-agnostic. To meet deadlines, administrators add cluster resources, use a capacity scheduler [96], or require users to reserve resources in advance [97, 98, 104]. Such solutions are costly, inefficient, or impractical, especially for resource constrained clusters.

Another issue encountered in multi-analytics systems, is that frameworks like Hadoop and Spark, which run on top of these resource allocators, have their own intra-job schedulers that greedily occupy the resources allocated to them, even when they are not using them [25, 79, 67]. *CARBYNE* [79] attempts to address this issue by exploiting task-level resource requirements information and DAG dependencies. It also uses prior runs of recurring jobs to estimate task demands and durations. Then, it intervenes both at the higher level, on the resource allocator, and internally, on framework task schedulers. It withholds a fraction of job resources from jobs that do not use them while maintain-

ing similar completion times. PYTHIA [21] is addressing the same issue by introducing framework-independent admission control that resource allocators can use to support dynamic fair-sharing of the cluster and deadline driven workflows for either Hadoop or Spark jobs, without requiring task-level information or depending on recurring jobs. Similar to PYTHIA (and contrary to CARBYNE), Pythia utilizes admission control without requiring job-repetitions and task-level information. Moreover, Pythia adapts to changing cluster conditions to avoid resource over-provisioning and preserves fair-sharing on top of satisfying deadlines. **Performance Prediction**: In order to allocate the required resources and meet job deadlines, much related work focuses on exploiting historic [17, 13, 83, 18, 84, 92, 19, 105], and runtime [14, 80, 17, 13, 14, 81, 82, 18, 92, 19] job information, while other research [94, 82, 85, 20, 78, 19] focuses on building job performance profiles and scalability models offline. Although, effective in many situations, we show that approaches similar to these suffer when used under resource constrained settings.

Strategies that depend solely on repeated jobs, by definition, do not guarantee performance of ad-hoc queries. While approaches that use runtime models, sampling, simulations, and extensive monitoring, impose overheads and additional costs. Moreover, trace analysis in this chapter and other research [20] shows that some production clusters have small ratio of repeated jobs and these jobs have often large execution times dispersion. Therefore, approaches based on past executions might not have the required mass of similar jobs over a short period of time in order to predict with high statistical confidence. Furthermore, the vast number of jobs have very short computation times [20, 45, 47, 74, 95]. Thus, approaches that adapt their initial allocation after a job has already started might be ineffective. Lastly, most of these approaches require task-level information, for the specific framework they target, either Hadoop [80, 94, 17, 13, 14, 81, 82, 83, 18, 84, 92, 85] or Spark [15, 16]. For this reason,

they cannot be used on top of resource managers like Mesos.

Justice in contrast, does not depend on job repetitions and can therefore target clusters with more diverse workloads; it does not impose overheads to perform extensive runtime monitoring or use job sampling and offline simulations to predict performance. *Justice* is also framework-independent because it does not require modeling of the different stages of any particular big data framework. Lastly, unlike all these approaches, *Justice* focuses on satisfying job deadlines in addition to preserving fair-sharing across jobs utilizing the cluster.

Admission Control: Admission control has been suggested as a solution for SaaS providers to effectively utilize their clusters and meet Service Level Agreements (SLAs) [99, 100], to provide map-reduce-as-a-service [101], and to resolve blocking caused by greedy YARN allocations [67]. *Justice* is similar in that but targets multi-analytics, resource-constrained clusters. We design *Justice* for use by resource managers for deadline-driven big data workloads, to be framework and task independent.

5.6 Summary

In this chapter, we present *Justice*, a fair-share and deadline-aware resource allocator with admission control for multi-analytic cluster managers like Mesos and YARN. *Justice* uses historical job statistics and deadline information to automatically adapt its resource allocation and admission control mechanisms to changing workload conditions. By doing so, it is able to estimate the minimum number of resources to allocate to a job to meet its deadline "just-in-time". Thus, it utilizes resources efficiently and satisfies job deadlines while preserving fair-share.

We evaluate *Justice* using trace-based simulation of large production YARN workloads in resource-constrained settings and under different deadline formulations. We

compare *Justice* to the existing fair-share allocator that ships with Mesos and YARN and find that *Justice* outperforms it significantly in terms of fairness, deadline satisfaction and efficient resource usage. Unlike, fair-share allocators that, when resources are limited, are unable adapt their decisions and as a result, violate fairness and waste significant resources for jobs that miss their deadlines, *Justice* monitors the changing cluster conditions and applies admission control to minimize resource waste while preserving resource allocation fairness and meeting more deadlines.

Justice is a practical solution that can work on top of existing open-source resource managers like Mesos and YARN. Its predictions do not depend on the internal structure of the processing engines running on top of the resource manager (e.g., Hadoop, Spark) nor it interacts with them in any way. Therefore, it is easy to maintain as it does not need to adapt to the fast evolution of the processing engines but only to the API changes of the resource manager. Moreover, *Justice* does not rely on job repetitions which make it suitable even for generic workloads that include ad-hoc queries. *Justice* is ideal for the constrained IoT analytics settings, not only because it is optimized to avoid wasting resources for infeasible jobs but also because it does not perform offline simulations, sampling, or extensive online monitoring that would require more computing resources and additional overheads. Lastly, *Justice* does not add complexity, as it only requires minimal information (a deadline and the input size) from the resource manager and the user.

As future work, we are planning to extend *Justice* so it can deal with more diverse workloads. We want to consider applications with different deadline criticality, including hard and soft deadlines, but also applications without deadlines. Also, we want to consider infeasible deadlines, i.e., the cluster cannot meet the deadline the user specified even under ideal allocation, and find ways to incentivize users to assign realistic deadlines and penalize them when their deadlines are very tight. Moreover, we want to evaluate

our mechanism taking into consideration memory allocations in addition to CPU, as well as understand the performance of *Justice* for different types of applications (e.g., CPU Vs I/O bound). Currently, *Justice* mechanism does not depend on job repetitions and works well for generalized workloads like the ones we used in this paper. However, for workloads where recurrent jobs constitute a significant amount, exploiting job repetitions patterns might further improve allocation efficiency. Lastly, we want to evaluate *Justice* with IoT analytics workloads and deploy it in real IoT analytics clusters, so we can derive insights from more field-specific characteristics.

Chapter 6

Conclusion

Big data analytics systems, designed for very-large scale and fault-tolerant operation, have revolutionized data processing and boosted the growth of a variety of industries. Web-search, advertising, content management, financial, e-commerce, healthcare, education, social-networks, sports are just few of the sectors that take advantage of the fast analysis and prompt decision support that big data processing systems enable. These complex distributed systems were originally designed to deal with the scale of tech-giants like Google, Yahoo, Facebook, and Twitter and, consequently, they were deployed in very-large, resource-rich, and dedicated clusters.

The advent of IoT and a sequence of recent technological advances enables wider adoption and brings combined operation of big data processing systems in smaller, resource-constrained, and shared clusters. First, open-source frameworks of these systems, like Hadoop, Spark, and Storm, are now available to the development community and to anyone willing to use their power for data analysis. Second, the wide adoption of cloud computing, which offers low cost compute, storage, and networking, makes open-source big data deployments a cost-efficient solution for smaller-scale companies, scientists, and practitioners. Third, resource managers like Mesos and YARN, enable the combined

use of multiple processing frameworks on the same cluster, opening the road for more compact multi-analytics deployments. Last, but certainly not least, the proliferation of IoT, witch made inexpensive sensing devices available to everyone, leads to an explosive growth of collected data and consequently increases the demand for extracting actionable insights from it. Moreover, to allow for faster decision support and actuation, IoT data analysis is done near where data sensing and collection takes places; in edge computing systems, which are small to medium-sized clusters.

The advent of IoT brings combined operation of big data processing systems in smaller, resource-constrained, and shared clusters. The proliferation of inexpensive IoT sensing and monitoring devices available to everyone, leads to an explosive growth of collected data and consequently increases the demand for extracting actionable insights from it. Moreover, to enable faster decision support and actuation, IoT data analysis is done near where data sensing and collection takes places; in edge computing systems, which are small to medium-sized clusters, or compact private clouds. Suche deployments are possible due to a sequence of recent technological advances. First, big data analytics frameworks like Hadoop, Spark, and Storm, are open-source and available to everyone. Second, the wide adoption of cloud computing, which offers low cost compute and storage, makes such deployments a cost-efficient solution. Third, resource managers like Mesos and YARN, enable the combined use of multiple processing frameworks on the same cluster, opening the road for more compact multi-analytics deployments.

In such resource-constrained and shared clusters for IoT multi-analytics, were low-latency actuation is necessary, applications have specific performance objectives that drives the resource allocator design. That is, the resource allocator should allocate sufficient resources to applications so they can complete their execution before a specific deadline because completing after the deadline provides no value to the user. Thus, deadline-aware resource allocation policies must maximize the number of satisfied dead-

lines additionally to preserving fairness. Moreover, resource allocators designed specifically for processing systems that analyze IoT data should be suitable for edge computing settings. That is, they should be framework-agnostic so they can run on the resource-manager level, operate under resource-constrained settings without adding overheads, and preserve fairness under congestion. However, the existing resource allocators are not designed to meet deadlines in such settings.

This work aims at evaluating the effectiveness of current fair-share allocation policies in multi-analytic and resource-constrained clusters and exploring feasible allocation mechanisms to satisfy deadlines and preserve fairness. Considering that modern data processing pipelines host multiple complicated analytic systems under the management of a single resource-manager, we want to provide a solution that does not depend on framework specific task information or requires instrumentation of each analytic system, but it is, instead, suitable to operate on the resource manager level across all frameworks. Moreover, as we target resource-constrained clusters, we aim for a design that utilizes efficiently all the available cluster resources and predicts congestion without adding significant overheads or requiring expensive offline processing. Lastly, we aim for a generic solution that does not depend solely on specific cluster characteristics (e.g. job repetitions) or deadline types, so it can be suitable for real, production cluster settings. Meeting these goals will facilitate the proliferation of familiar, mature, open-source big data processing systems and their well developed ecosystem of tools and frameworks, in resource-constrained multi-analytic settings.

With the goal of preserving fairness and meeting deadlines in multi-analytics resource-constrained cluster settings in a framework-agnostic way, we:

1. Investigate and characterize the performance and behavior of big/fast data systems in multi-tenant, resource constrained settings using popular, open-source frame-

works.

2. Design and empirically evaluate an admission control strategy for resource managers that manage deadline-driven workloads in resource-constrained cluster settings.

3. Investigate the efficacy of fair-share schedulers in resource-constrained cluster settings. Design and empirically evaluate a resource allocator that uses deadline information and historical workload analysis to adapt its allocations on the changing cluster conditions and use resources efficiently to achieve fairness and satisfy job deadlines.

To understand how popular analytics frameworks behave and interfere under resource constrained settings, we investigate the performance and behavior of distributed batch and stream processing systems that share resource constrained, private clouds managed by the Mesos resource manager [25]. We examine how these systems interfere with each other and Mesos, to evaluate the effect on systems performance, overhead, and fair resource sharing. Our experimental results show that in constrained environments, there is significant performance interference that manifests in multiple ways. First, Mesos is not always able to preserve fairness. Moreover, application performance depends on Mesos offer order and is highly variable. Finally, we find that resource allocation among tenants with different scheduling designs, can lead to starvation, deadlocks, and underutilization of system resources.

To satisfy deadlines and utilize the cluster productively in resource constrained multi-analytic settings, we design and built Pythia [21]. Pythia is framework-agnostic and as such, suitable for resource managers for big data multi-analytics systems, such as Apache Mesos and YARN. Pythia adds support for deadlines and facilitates more effective resource use when resources are constrained due to physical limitations (private

clouds) and cost contraints (when using public clouds), two increasingly common big data deployment scenarios.

Pythia builds a cluster-wide model based on historical job runs and deadlines to estimate the minimum amount of resources an admitted job needs to meet its deadline. If these resources are not available, Pythia drops the job to avoid wasting resources for jobs with infeasible deadlines. If the resources exist, the job is accepted and runs until completion unless it exceeds its deadline. Our experiments with real-world traces show that Pythia meets significantly more deadlines than existing fair-share approaches and eliminates resource-waste in resource constrained clusters.

Finally, to preserve fairness under resource constrained-resources while meeting job deadlines, we design *Justice* [106]. *Justice* is a fair-share and deadline-aware resource allocator suitable for big data resource managers. *Justice* uses deadline information and historical job execution logs and adapts to changing cluster conditions, to assign enough resources for each job to meet its deadline "just-in-time".

Justice tracks the success of its allocation decisions and improves its future allocations accordingly. Every time a job completes, it updates a cluster-wide model that includes information about the duration, size, maximum parallelization, deadline, and provided resources for each job. If the job is successful, *Justice* becomes more optimistic providing the jobs that follow with less resources hoping they will still meet their deadlines. In contrast, if the job is unsuccessful, *Justice* provides more conservative allocations to make sure no more jobs miss their deadlines. The results, from trace-based simulation on large production YARN workloads, show that *Justice*, in resource-constrained settings, outperforms the existing fair-share allocation policy implemented on resource-managers like YARN and Mesos in terms of fairness, deadline satisfaction, and efficient resource usage.

Our research aims to satisfy deadlines and preserve fairness to enable reliable use of

multi-analytic systems in resource constrained clusters. It achieves this in a frameworks-agnostic way, by utilizing admission control and predicting resource requirements without exploiting job repetitions. Thus, it can be used on top of resource managers like Apache Mesos, in a pluggable way, without requiring extensive modifications of the manager or the frameworks running on top. Lastly, our research solutions are ideal for resource-limited clusters, like the ones used for IoT data analysis, because it does not add processing overheads and does not perform extensive monitoring that would waste precious cluster resources.

A key point of our research is its applicability, without costly modifications and maintenance, in existing, popular open-source systems like Apache Mesos and YARN. Our approach exploits information available on the resource manager level (e.g., Mesos) and does not require integration with the variety of diverse and fast evolving processing engines (e.g., Hadoop, Spark) that run on top. Thus, it requires minimal effort to integrate with the resource manager without the need to adapt to API or structural changes of the processing engines.

Bibliography

[1] "Apache Hadoop." `https://hadoop.apache.org/`. [Online; accessed 21-January-2016].

[2] "Apache Spark." `http://spark.apache.org/`. [Online; accessed 21-January-2016].

[3] "Apache Storm." `http://storm.apache.org/`. [Online; accessed 21-January-2016].

[4] "Storm Users." `http://storm.apache.org/documentation/Powered-By.html`. [Online; accessed 27-January-2016].

[5] "Applications Powered By Samza." `https://cwiki.apache.org/confluence/display/SAMZA/Powered+By`. [Online; accessed 9-May-2017].

[6] "Hadoop Users." `http://wiki.apache.org/hadoop/PoweredBy`. [Online; accessed 27-January-2016].

[7] "Spark Users." `https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark`. [Online; accessed 27-January-2016].

[8] "HDFS Architecture Guide." `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. [Online; accessed 21-January-2016].

[9] 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 *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

[10] "Apache Mesos." `http://mesos.apache.org/`. [Online; accessed 21-January-2016].

[11] "YARN Fair Scheduler." `https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`. [Online; accessed 4-January-2017].

[12] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, *Dominant resource fairness: Fair allocation of multiple resource types.*, in *NSDI*, vol. 11, pp. 24–24, 2011.

[13] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Campbell, *Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle*, in *2012 IEEE Network Operations and Management Symposium*, pp. 900–905, IEEE, 2012.

[14] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, *Rush: A robust scheduler to manage uncertain completion-times in shared clouds*, in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 242–251, IEEE, 2016.

[15] S. Sidhanta, W. Golab, and S. Mukhopadhyay, *Optex: A deadline-aware cost optimization model for spark*, *arXiv preprint arXiv:1603.07936* (2016).

[16] K. Wang and M. M. H. Khan, *Performance prediction for apache spark platform*, in *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, pp. 166–173, IEEE, 2015.

[17] A. Verma, L. Cherkasova, and R. H. Campbell, *Aria: automatic resource inference and allocation for mapreduce environments*, in *ACM International Conference on Autonomic Computing*, pp. 235–244, 2011.

[18] M. Hu, C. Wang, P. You, Z. Huang, and Y. Peng, *Deadline-oriented task scheduling for mapreduce environments*, in *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 359–372, Springer, 2015.

[19] C. Delimitrou and C. Kozyrakis, *Quasar: resource-efficient and qos-aware cluster management*, *ACM SIGPLAN Notices* **49** (2014), no. 4 127–144.

[20] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, *Jockey: guaranteed job latency in data parallel clusters*, in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 99–112, ACM, 2012.

[21] S. Dimopoulos, C. Krintz, and R. Wolski, *Pythia: Admission control for multi-framework, deadline-driven, big data workloads*, in *International Conference on Cloud Computing*, IEEE, 2017.

[22] "Apache Mahout." `http://mahout.apache.org/`. [Online; accessed 25-November-2015].

[23] "Spark MLlib." `https://spark.apache.org/docs/1.1.0/mllib-guide.html`. [Online; accessed 25-November-2015].

[24] "Tensor Flow." `https://www.tensorflow.org/`. [Online; accessed 25-November-2015].

[25] S. Dimopoulos, C. Krintz, and R. Wolski, *Big data framework interference in restricted private cloud settings*, in *IEEE International Conference on Big Data*, IEEE, 2016.

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, *Spark: cluster computing with working sets*, in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, p. 10, 2010.

[27] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, *Apache flink: Stream and batch processing in a single engine*, *Data Engineering* (2015) 28.

[28] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, *Naiad: a timely dataflow system*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, ACM, 2013.

[29] "Cloudera." `https://www.cloudera.com/`. [Online; accessed 25-November-2015].

[30] "Horton Works." `https://hortonworks.com/`. [Online; accessed 25-November-2015].

[31] "MapR." `https://mapr.com/`. [Online; accessed 25-November-2015].

[32] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, *Musketeer: all for one, one for all in data processing systems*, in *Proceedings of the Tenth European Conference on Computer Systems*, p. 2, ACM, 2015.

[33] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal, *Optimizing analytic data flows for multiple execution engines*, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 829–840, ACM, 2012.

[34] K. Doka, N. Papailiou, V. Giannakouris, D. Tsoumakos, and N. Koziris, *Mix'n'match multi-engine analytics*, in *IEEE International Conference on Big Data. IEEE*, 2016.

[35] S. Cheng, J. A. Stankovic, and K. Ramamritham, *Scheduling algorithms for hard real-time systems–a brief survey*, .

[36] T. L. Casavant and J. G. Kuhl, *A taxonomy of scheduling in general-purpose distributed computing systems*, *IEEE Transactions on software engineering* **14** (1988), no. 2 141–154.

[37] N. Audsley and A. Burns, *Real-time system scheduling*, .

[38] A. Burns, *Scheduling hard real-time systems: a review*, *Software Engineering Journal* **6** (1991), no. 3 116–128.

[39] H. Chetto, M. Silly, and T. Bouchentouf, *Dynamic scheduling of real-time tasks under precedence constraints*, *Real-Time Systems* **2** (1990), no. 3 181–194.

[40] C. L. Liu and J. W. Layland, *Scheduling algorithms for multiprogramming in a hard-real-time environment*, *Journal of the ACM (JACM)* **20** (1973), no. 1 46–61.

[41] K. Ramamritham and J. A. Stankovic, *Dynamic task scheduling in hard real-time distributed systems*, *IEEE software* **1** (1984), no. 3 65.

[42] K. Schwan, P. Gopinath, and W. Bo, *Chaos-kernel support for objects in the real-time domain*, *IEEE Transactions on computers* **100** (1987), no. 8 904–916.

[43] L. Sha, J. P. Lehoczky, and R. Rajkumar, *Task scheduling in distributed real-time systems*, in *Robotics and IECON'87 Conferences*, pp. 909–917, International Society for Optics and Photonics, 1987.

[44] R. Chipalkatti, J. Jurose, and D. Towsley, *Scheduling policies for real-time and non-real-time traffic in a statistical multiplexer*, in *INFOCOM'89. Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies. Technology: Emerging or Converging, IEEE*, pp. 774–783, IEEE, 1989.

[45] Y. Chen, S. Alspaugh, and R. Katz, *Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads*, *Proceedings of the VLDB Endowment* **5** (2012), no. 12 1802–1813.

[46] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou, *Workload characterization on a production hadoop cluster: A case study on taobao*, in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pp. 3–13, IEEE, 2012.

[47] K. Ren, G. Gibson, Y. Kwon, M. Balazinska, and B. Howe, *Hadoop's adolescence; a comparative workloads analysis from three research clusters.*, in *SC Companion*, p. 1452, 2012.

[48] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, *Mesos: A platform for fine-grained resource sharing in the data center.*, in *NSDI*, vol. 11, pp. 22–22, 2011.

[49] 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 *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[50] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, *Clash of the titans: Mapreduce vs. spark for large scale data analytics*, *Proceedings of the VLDB Endowment* **8** (2015), no. 13 2110–2121.

[51] F. Liang, C. Feng, X. Lu, and Z. Xu, *Performance benefits of datampi: a case study with bigdatabench*, in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pp. 111–123. Springer, 2014.

[52] "Mesos Roles." `http://mesos.apache.org/documentation/latest/roles/`. [Online; accessed 28-December-2016].

[53] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, *The eucalyptus open-source cloud-computing system*, in *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pp. 124–131, IEEE, 2009.

[54] "Apache Mesos 0.21.1 Release." `http://mesos.apache.org/blog/mesos-0-21-1-released/`. [Online; accessed 1-December-2015].

[55] "CDH 5.1.2 Release." `http://www.cloudera.com/content/www/en-us/downloads/cdh/5-1-2.html`. [Online; accessed 1-December-2015].

[56] "Apache Spark 1.2.1 Release." `https://spark.apache.org/releases/spark-release-1-2-1.html`. [Online; accessed 1-December-2015].

[57] "Apache Storm 0.9.2 Release." `http://storm.apache.org/2014/06/25/storm092-released.html`. [Online; accessed 1-December-2015].

[58] "Bug Fixes for Hadoop On Mesos." `https://github.com/strat0sphere/hadoop`. [Online; accessed 1-December-2015].

[59] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, *et. al.*, *Bigdatabench: A big data benchmark suite from internet services*, in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 488–499, IEEE, 2014.

[60] "Modifications on Benchmarking Code." `https://github.com/MAYHEM-Lab/benchmarking-code`. [Online; accessed 22-January-2015].

[61] M. L. Massie, B. N. Chun, and D. E. Culler, *The ganglia distributed monitoring system: design, implementation, and experience*, *Parallel Computing* **30** (2004), no. 7 817–840.

[62] "Lambda Architecture." `http://lambda-architecture.net/`. [Online; accessed 1-December-2015].

[63] "Twitter Summingbird." `https://github.com/twitter/summingbird`. [Online; accessed 27-January-2016].

[64] "Lambdoop." `https://novelti.io/lambdoop/`. [Online; accessed 27-January-2016].

[65] "Lambda on Metamarkets." `https://metamarkets.com/2014/building-a-data-pipeline-that-handles-billions-of/-events-in-real-time/`. [Online; accessed 27-January-2016].

[66] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et. al.*, *Storm@ twitter*, in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156, ACM, 2014.

[67] Y. Yao, J. Lin, J. Wang, N. Mi, and B. Sheng, *Admission control in yarn clusters based on dynamic resource reservation*, in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 838–841, IEEE, 2015.

[68] L. Gu and H. Li, *Memory or time: Performance evaluation for iterative operation on hadoop and spark*, in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pp. 721–727, IEEE, 2013.

[69] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI, *Making sense of performance in data analytics frameworks*, in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)(Oakland, CA*, pp. 293–307, 2015.

[70] B. Li, Y. Diao, and P. Shenoy, *Supporting scalable analytics with latency constraints*, *Proceedings of the VLDB Endowment* **8** (2015), no. 11 1166–1177.

[71] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, *A platform for scalable one-pass analytics using mapreduce*, in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 985–996, ACM, 2011.

[72] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, *Discretized streams: Fault-tolerant streaming computation at scale*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 423–438, ACM, 2013.

[73] J. Dean and S. Ghemawat, *Mapreduce: simplified data processing on large clusters*, *Communications of the ACM* **51** (2008), no. 1 107–113.

[74] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, *A comparison of approaches to large-scale data analysis*, in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 165–178, ACM, 2009.

[75] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, *Woha: deadline-aware map-reduce workflow scheduling framework over hadoop clusters*, in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pp. 93–103, IEEE, 2014.

[76] K. Kc and K. Anyanwu, *Scheduling hadoop jobs to meet deadlines*, in *International Conference on Cloud Computing*, pp. 388–392, 2010.

[77] R. Sumbaly, J. Kreps, and S. Shah, *The big data ecosystem at linkedin*, in *ACM SIGMOD International Conference on Management of Data*, pp. 1125–1134, 2013.

[78] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, *Ernest: efficient performance prediction for large-scale advanced analytics*, in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 363–378, 2016.

[79] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, *Altruistic scheduling in multi-resource clusters*, in *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[80] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguade, M. Steinder, and I. Whalley, *Performance-driven Task Co-Scheduling for MapReduce Environments*, in *IEEE Network Operations and Management Symposium*, pp. 373–380, 2010.

[81] H. Herodotou and S. Babu, *Profiling, what-if analysis, and cost-based optimization of mapreduce programs*, *VLDB* **4** (2011), no. 11 1111–1122.

[82] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu, *Mimp: Deadline and interference aware scheduling of hadoop virtual machines*, in *IEEE Cluster, Cloud and Grid Computing*, pp. 394–403, 2014.

[83] P. Lama and X. Zhou, *Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud*, in *ACM International Conference on Autonomic Computing*, pp. 63–72, 2012.

[84] N. Zaheilas and V. Kalogeraki, *Real-time scheduling of skewed mapreduce jobs in heterogeneous environments*, in *11th International Conference on Autonomic Computing (ICAC 14)*, pp. 189–200, 2014.

[85] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, *Bridging the tenant-provider gap in cloud services*, in *ACM Symposium on Cloud Computing*, 2012.

[86] "Simpy." `https://simpy.readthedocs.io/en/latest/`. [Online; accessed 4-January-2017].

[87] A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica, *Hierarchical scheduling for diverse datacenter workloads*, in *ACM SoCC*, 2013.

[88] E. Friedman, A. Ghodsi, and C.-A. Psomas, *Strategyproof allocation of discrete jobs on multiple machines*, in *ACM EC*, 2014.

[89] J. Tan, L. Zhang, M. Li, and Y. Wang, *Multi-resource fair sharing for multiclass workflows*, *ACM SIGMETRICS Performance Evaluation Review* **42** (2015), no. 4.

[90] W. Wang, B. Liang, and B. Li, *Multi-resource fair allocation in heterogeneous cloud computing systems*, *IEEE Trans. Parallel Distrib. Syst.* **26** (2015), no. 10.

[91] J. Liu, H. Shen, and H. S. Narman, *Ccrp: Customized cooperative resource provisioning for high resource utilization in clouds*, in *IEEE International Conference on Big Data. IEEE*, 2016.

[92] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo, *Automated profiling and resource management of pig programs for meeting service level objectives*, in *ACM International Conference on Autonomic computing*, pp. 53–62, 2012.

[93] "Mesos Modules." `http://mesos.apache.org/documentation/latest/allocation-module/`. [Online; accessed 4-January-2017].

[94] F. Tian and K. Chen, *Towards optimal resource provisioning for running mapreduce programs in public clouds*, in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pp. 155–162, IEEE, 2011.

[95] J. Liu, H. Shen, and H. S. Narman, *Ccrp: Customized cooperative resource provisioning for high resource utilization in clouds*, *Resource* **40** (2016) 60.

[96] "YARN Capacity Scheduler." `https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`. [Online; accessed 4-January-2017].

[97] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, *Reservation-based scheduling: If you're late don't blame us!*, in *ACM Symposium on Cloud Computing*, pp. 1–14, 2014.

[98] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, *Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters*, in *European Conference on Computer Systems*, p. 35, 2016.

[99] L. Wu, S. K. Garg, and R. Buyya, *Sla-based admission control for a software-as-a-service provider in cloud computing environments*, *Journal of Computer and System Sciences* **78** (2012), no. 5 1280–1299.

[100] M. Islam, P. Balaji, P. Sadayappan, and D. K. Panda, *Towards provision of quality of service guarantees in job scheduling*, in *IEEE International Conference Cluster Computing*, IEEE, 2004.

[101] J. Dhok, N. Maheshwari, and V. Varma, *Learning based opportunistic admission control algorithm for mapreduce as a service*, in *India software engineering conference*, pp. 153–160, ACM, 2010.

[102] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*, vol. 38. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.

[103] "Welch's T-Test." `https://en.wikipedia.org/wiki/Welch's_t-test`. [Online; accessed 22-Jul-2017].

[104] M. Babaioff, Y. Mansour, N. Nisan, G. Noti, C. Curino, N. Ganapathy, I. Menache, O. Reingold, M. Tennenholtz, and E. Timnat, *Era: A framework for economic resource allocation for the cloud*, in *Proceedings of the 26th International Conference on World Wide Web Companion*, pp. 635–642, International World Wide Web Conferences Steering Committee, 2017.

[105] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, *Morpheus: towards automated slos for enterprise clusters*, in *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, p. 117, 2016.

[106] S. Dimopoulos, C. Krintz, and R. Wolski, *Justice: A deadline-aware, fair-share resource allocator for implementing multi-analytics*, in *Cluster Conference*, IEEE, 2017.