Machine Learning for Automatic Resource Management in the Datacenter and the Cloud

by

Neeraja Jayant Yadwadkar

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

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Summer 2018

Machine Learning for Automatic Resource Management in the Datacenter and the Cloud

Abstract

Machine Learning for Automatic Resource Management in the Datacenter and the Cloud

by

Neeraja Jayant Yadwadkar

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy H. Katz, Co-chair

Assistant Professor Joseph E. Gonzalez, Co-chair

Traditional resource management techniques that rely on simple heuristics often fail to achieve predictable performance in contemporary complex systems that span physical servers, virtual servers, private and/or public clouds. My research aims to bring the benefits of data-driven models to resource management of such complex systems. In my dissertation, I argue that the advancements in machine learning can be leveraged to manage and optimize today's systems by deriving actionable insights from the performance and utilization data these systems generate. To realize this vision of model-based resource management, we need to deal with the key challenges data-driven models raise: uncertainty in predictions, cost of training, generalizability from benchmark datasets to real-world systems datasets, and interpretability of the models.

In this dissertation, to demonstrate how to handle these challenges, we chose two main problem domains: (I) Scheduling in parallel data intensive computational frameworks for improved tail latencies, and (II) Performance-aware resource allocation in the public cloud environments for meeting user-specified performance and cost goals.

We begin by presenting Wrangler, a system that predicts when stragglers (slow-running tasks) are going to occur based on cluster resource utilization counters and makes scheduling decisions to avoid such situations. Wrangler introduces a notion of a confidence measure with these predictions to overcome modeling uncertainty. We then describe our Multi-Task Learning formulations that share information between the various models, allowing us to significantly reduce the cost of training. To capture the challenges of resource allocation in the public cloud environments, we present key observations from our empirical analysis based on performance profiles of workloads executing across different public cloud environments. Finally, we describe PARIS, a Performance-Aware Resource Inference System, that we built to enable cloud users to select the best VM (virtual machine) for their applications in the public cloud environments so as to satisfy any performance and cost constraints.

To my husband, Vivek Chawda.
To my family, especially my parents, Vasudha Yadwadkar and Jayant Yadwadkar.
To all my teachers, especially Prof. Chiranjib Bhattacharyya.
To everyone who has believed in me.

Contents

List of Figures

List of Tables

Acknowledgments

I want to express my deepest gratitude to my advisor, Randy Katz, for teaching me how to do good research, for always believing in me, and for encouraging me throughout my grad life. Randy has been a role model for me in many ways and has taught me how to be calm in stressful situations, how to care for and cater to the needs of students, and most importantly, how to be disciplined. I am grateful to his relentless efforts in helping me improve my writing skills by marking my paper drafts thoroughly multiple times. I will always value his guidance, and gracious efforts to provide all the tools, technical and financial, that allowed me to focus on research better. Thanks also to my co-advisor, Joseph Gonzalez, for mentoring me. His push for thorough empirical analysis coupled with theoretical soundness has helped my research tremendously. I also want to thank him for helping me develop my presentation skills to give effective technical talks.

I would also like to thank my dissertation committee members, Vern Paxson, Ken Goldberg, and Burton Smith, for their valuable feedback all through my qualification exam and dissertation. Vern's thorough comments and hard questions have shaped the writing and empirical data analysis in this dissertation greatly. I am thankful to Vern for inspiring me to be curious and skeptical about what the representation chosen for data might be hiding from us. Thanks to Ken for always encouraging me, and for showing confidence in me. I would also like to thank Burton who I had an opportunity to work closely with while interning at Microsoft Research. Burton, unfortunately, passed away recently. I am grateful to him for showing me how to focus on the fundamentals, and how to design systems from the first principles. Thanks also to my prelim committee members, David Culler and John Kubiatowicz, for providing me with a great exam that helped me feel ready for systems research.

I feel very fortunate to have Bharath Hariharan as a close friend as well as a mentor. He has always been ready to help with many questions I have related to research and life in general. I also want to thank Ganesh Ananthanarayanan, who I worked with on my very first project as a grad student. From Bharath and Ganesh, I learned how to write from the reader's perspective, how to explain complex things well, and how to persist through hard research times. I am extremely thankful to have found a great academic elder brother in Yanpei Chen. His perpetual readiness for helping me and advising me continues to amaze me. John Wilkes has played the role of a critical mentor whose inputs have shaped this work tremendously. His feedback on my posters and talks has helped me improve my presentation skills. Marvin Theimer's industrial perspective on my work has guided my work to be more relevant and applicable in real world.

I had the fortune to work with Anthony Joseph and Ion Stoica as a graduate student instructor (GSI) for the undergrad operating systems class for two semesters at Berkeley. I learned a lot from both of them. Ion gave me the opportunity to teach a couple of lectures which further reinforced my passion for teaching.

I also want to thank all the teachers with whom I have taken classes at Berkeley: Martin Wainwright, John Kubiatowicz, Anthony Joseph, Sylvia Ratnasamy, Ion Stoica, Vern Paxson, Laurent El Ghaoui, Raluca Popa, Joe Hellerstein, and Zachary Pardos.

I want to thank all my friends and colleagues in the AMPLab, RISELab, NetSys, and ParLab, who were always available for long brainstorming sessions or quick questions: Sarah Chasins, Mangpo, Sarah Bird, Xinghao, Prashant, David, Sara Alspaugh, Yanpei, Wontae, Daniel Haas, Purnamrita, Sameer, Colin, Panda, Shivaram, TD, Charles, Virginia, Evan, Andre, Anurag, Anand, Xin, Wenting, Gulio, Eugene, Alyssa, Rashmi, Elaine, Kaifei, Gautam, Aishwarya, Orianna, Qifan, Vikram, Rolando, Moustafa, Jack, and Dan Crankshaw. Many thanks to each one of you for helping me in many ways.

I would like to express my heartfelt gratitude to the incredible people who worked hard in many different ways to enrich my grad school experience - Shirley Salanio, Kattt Atchley, Boban Zarkovich, Carlyn Chinen, Jon Kuroda, Shane Knapp, Albert Goto, Matt Massie, and Audrey Sillers.

I am truly thankful to the energy and support I have received over the last 5 years from my dance teacher, Seibi Lee, and my dance sisters, Aradhana Vaidya, and Ria DasGupta.

I am extremely thankful for the fun and support I found through the close circle of friends at Berkeley: Bharath, Shubham, Yanpei, Gireeja, Purnamrita, Rashmi, Nihar, Archana, Anand, Lakshmi, Nikunj, Sreeta, Nitika, Arnav, Gautam, Anurag, Shromona, Divya, Xin, Wenting, Radhika, Saurabh, TD, Nitesh, and Aishwarya. Thanks to everyone for always being ready to help out.

I feel fortunate to have studied with Chiranjib Bhattacharyya, my master's advisor, who introduced me to research. I am extremely grateful to him for preparing me for this wonderful journey and for continuously nourishing me technically and emotionally throughout these years.

Finally, I feel fortunate to have the most loving and supportive family who always encouraged me and were always there when I needed them the most. My parents, Vasudha Yadwadkar and Jayant Yadwadkar, have given their lives to ensure I have the most comfortable life. They instilled in me the value of being a good person. I also learned the importance of hard work, persistence and positivity by observing them over the years. I also thank my parents-in law for supporting me over the years. I am grateful to have found a close friend in my brother-in-law, Vaibhav Chawda, who is always ready to give a listening and thoughtful ear to everything I have to say. Finally, and most importantly, I think the person who made this dissertation feasible, is my best friend and husband, Vivek Chawda. I am eternally thankful to Vivek for his firm belief in me even in the moments I doubted myself. Thanks to Vivek for giving countless days and nights to support my grad life, and for commuting over 100 miles every single day over 5 years just to ensure I have the most comfortable stay and commute to focus on my work.

Chapter 1

Introduction

In the past couple of decades, we have seen data getting generated at an unprecedented rate. This growth can be attributed mainly to ubiquitous devices and applications. Ubiquitous devices, such as security cameras, handheld scanners, tablets, wireless sensors give rise to enormous amount of data. Applications, including social media such as Facebook, Twitter, and the aggregations of such data with third party information, generate and send significant amount of data across the globe over the Internet. We also see emerging new applications enabled by such huge amount of data, for example, Siri or Cortana using audio data or home security applications using live feed of video data. These emerging applications have dictated the need for processing this data in realtime for unraveling important insights from them. For instance, analyzing a stream of live video data for any suspicious activity with an intention of generating an alarm and taking appropriate safety measures.

In 1965, Gordon Moore (co-founder of Intel) predicted that the transistor density of semiconductor chips would double roughly every eighteen months, called Moore's law [110, 133]. This has enabled the processing speed of a single core to scale rapidly. Storage and network capacity also increased exponentially over years. This allowed us to store enormous amount of data generated by the devices and applications. The increasing resource efficiency enabled personal computers and even small handheld devices to process large amounts of data. System designers and application developers have benefitted from this advancement in hardware resources resulting in performance improvement of applications without having to rewrite the software. However, Moore's law has ended [137]. This emphasizes the need for smarter resource management techniques that enable systems to deal with the fast growing data using the existing resources more efficiently. This dissertation aims at developing **automatic resource management** mechanisms for the datacenter and the cloud that enable distributed systems to achieve **faster** and **predictable performance** while **reducing the cost**.

1.1 Contemporary computing systems and challenges to resource management

With the decline of Moore's law and continuous growth in data, traditional algorithms are now largely distributed. These algorithms and applications that process huge volume of this data have caused two major changes in computing: (a) parallel execution of data intensive computations and (b) use of shared commodity hardware. Frameworks such as MapReduce [56], Spark [171], Naiad [113], and various distributed graph processing systems [74, 105] demonstrate the benefits of such distributed data processing mechanisms. These frameworks primarily run on economically available commodity hardware to offer various services such as elasticity, and scalability, to users at reduced cost. The advent of cloud computing, where instead of buying physical servers users can easily receive an on-demand delivery of compute, storage, and other resources for a short duration, has played a key role in the wide adoption of such distributed systems. Cloud computing providers such as AWS [8], Microsoft Azure [102], and Google Cloud Engine [97] allow users to benefit from economy of scale in terms of pay-as-you-go pricing model resulting in low overall costs for users.

Contemporary systems that support these changes in computing paradigms have started spanning physical servers, virtual servers, private and/or public clouds or a combination of these different resources. Evolution in computing has taken us from physical servers in datacenters to virtual servers in datacenters, to virtual servers in the cloud, to containers, and now functions run as a service in the cloud. With this trend we see increasing automation in provisioning for resources, making it feasible for programmers to focus more on application development without needing to think about provisioning and scaling of resources.

However, contemporary systems that span such a wide range of resource infrastructure present unique challenges to resource management especially when there is a pressing need for faster and predictable computation while optimizing the utilization of resources. To reduce the cost of applications running at a *large scale*, systems use inexpensive *commodity hardware* and *share* it across different users and different workloads. However, use of commodity hardware makes these systems failure-prone and sharing of resources gives rise to highly variable environments. This makes reasoning about performance challenging, which in turn makes resource management decisions such as where to assign or execute a job unclear. The virtualized public cloud environments present the challenge of *lack of complete control* of underlying hardware resources to their users. The complexity of these systems is further emphasized due to *heterogeneity across nodes and user workloads* running on them that adds to the problem of reasoning about performance and taking suitable resource management decisions. Thus, estimating resource requirements of workloads and the impact of selected resources on performance and mainly on variability in performance has become more challenging.

Traditional approaches to designing resource management solutions have been primarily driven by simple heuristics that capture *static* relationships between performance and resources. However, due to the challenges described earlier, including use of heterogeneous, shared commodity hardware, these relationships are likely to change *dynamically*. Heuristics that do not utilize the rich context available in the form of performance counters are unable to react to changing environments, causing either poor and unpredictable performance or poorly utilized resources.

In this dissertation, rather than relying on traditional heuristic-based or statically defined approaches, we propose to inform resource management decisions, such as scheduling or resource allocation, using *data-driven models*: models that leverage dynamically collected performance data. We note that such data is easily available today with systems capturing it regularly [86, 171, 155, 107] using various performance monitoring tools [122, 60, 107, 78]. Given that the large-scale distributed computing systems today show stochasticity in behavior [55], a better approach could be to leverage such data to get informed about the system state. In particular, we show that data-driven learning techniques enable more efficient and cost-effective solutions to optimizing performance and cost of computations in the datacenters and the public cloud environments than state-of-the-art mechanisms.

1.2 Vision: Data-Driven Models for resource management

We focus on the opportunity presented by the data collected regularly by contemporary systems. This data contains performance and utilization indicators; the insights derived from this data have the potential to impact various aspects of distributed systems, including resource management.

We exploit the opportunity presented by such data and propose an alternative approach to resource management for datacenters and other cloud-hosted systems. The key idea is to let the data itself drive resource management decisions augmenting existing approaches [86, 151, 134] based on expert knowledge, heuristics and simple modeling mechanisms. However, data is only as important as the actions it enables. To enable important decisions, we need to extract actionable insights from this data. We leverage the advances in the field of statistical learning and machine learning (ML) to build data-driven models. Such data-driven models can enable distributed systems to make decisions by automatically learning and adapting to dynamically changing state of the underlying system components.

However, we realize that off-the-shelf learning techniques may not be directly applicable to enable data-driven actions. As opposed to improving accuracy of predictions on test data, we desire to use the predictions from our models to influence resource management decisions taken. That raises certain challenges:

1. **Prediction error or model's uncertainty:** The impact of wrong predictions could be high. For instance, a wrong scheduling decision might cause violation of Service Level Objective (SLO) and may incur monetary penalties.

2. **Cost of training:** To train these data-driven models, we need to collect sufficient amount of training data. The cost in terms of time and/or money spent on collecting such data could be non-trivial.

3. **Generalization:** To enable adoption of the models in real system deployments, we need to ensure that we build models that can generalize from benchmark workloads to real user workloads.

4. **Interpretability:** When applying ML models for deriving important decisions in the domain of distributed systems, in addition to accurate predictions, it is helpful for domain experts to gain insights into the decision making process of the models.

To demonstrate how to handle these challenges, we chose two main problem domains: (I) Scheduling in parallel data intensive computational frameworks for improved tail latencies with reduced resource consumption (Chapters 3 and 4), and (II) Performance-aware resource allocation in the public cloud environments for meeting user-specified performance and cost goals (Chapter 5).

1.3 Overview of the dissertation

This dissertation aims at developing automatic resource management mechanisms that enable distributed systems to achieve faster and predictable performance while reducing cost. To achieve this goal, we develop practical model-based solutions guided by insights drawn from data generated by existing deployments. In the context of the two problem domains described in the previous section, below we outline the systems we designed, implemented, and evaluated.

1.3.1 Predictive Scheduling for parallel data intensive computational frameworks

Parallel processing frameworks [56] accelerate jobs by breaking them into tasks that execute in parallel. However, slow running or *straggler* tasks can run up to 8 times slower than the median task on a production cluster [11], leading to delayed job completion and inefficient use of resources. Existing straggler mitigation techniques wait to detect stragglers and then relaunch them, delaying straggler detection and wasting resources. We built Wrangler [160], a system that predicts when stragglers are going to occur and makes scheduling decisions to avoid such situations.

Building data-driven models requires collection of substantial training data. This data-collection phase could be time-consuming. To this end, we propose multi-task learning formulations that share information between the various models, allowing us to use less training data and bring training time down significantly. Unlike naive multi-task learning formulations, our formulations capture the shared structure in our data, improving generalization performance on limited data. Finally, we extend these formulations using group sparsity inducing norms [21] to automatically discover the similarities between models and improve interpretability.

1.3.2 Resource allocation in the public cloud environments

As companies of all sizes migrate to cloud environments, increasingly diverse workloads are being run in the Cloud — each with different performance requirements and cost trade-offs [129]. Recognizing this diversity, cloud providers offer a wide range of Virtual Machine (VM) types. For instance, at the time of writing, Amazon [6], Google [76], and Azure [102] offered a combined total of over 100 instance types with varying system and network configurations.

In this work, we address the fundamental problem of *accurately* and *economically* choosing the *best VM* for a given *workload* and *user goals*. This choice is critical because of its impact on performance metrics such as runtime, latency, throughput, cost, and availability. Yet determining or even defining the *"best"* VM depends heavily on the users' goals which may involve diverse, application-specific performance metrics, and span tradeoffs between price and performance objectives. We built PARIS, a data-driven system that uses a novel hybrid offline and online data collection and modeling framework to provide accurate performance estimates with minimal data collection. PARIS is able to predict workload performance for different user-specified metrics, and resulting costs for a wide range of VM types and workloads across multiple cloud providers.

1.3.3 Evaluation Methodology

The results and insights presented in this dissertation are derived from synthetic as well as real-world production-level workloads.

Production-level real-world workloads: We use job execution traces of data intensive computational frameworks from production clusters at Facebook and customers of Cloudera. All these production clusters support business critical applications. The traces describe a multi-job execution on underlying resources. These traces contain job-level statistics, such as time of submission, input/shuffle/output data sizes (bytes), duration, number of constituent tasks, and task-level durations. More details of these traces, and other workloads used in this dissertation are provided in corresponding Chapters 3 and 4.

Trace-driven Replay To evaluate using the production-level real-world traces on AWS, we use a trace-driven replay mechanism, called SWIM [44]. SWIM is a statistical workload replay tool that performs a detailed and faithful replay of the workload traces from Facebook and customers of Cloudera. SWIM synthesizes a workload with representative job submission rate and patterns, shuffle/input data size and output/shuffle data ratios (see [44] for details of replay methodology). SWIM scales the workload to the number of nodes in an experimental cluster. More details on the datasets collected through this trace-driven replay are provided in Chapters 3 and 4 along with their analysis and results obtained.

Benchmark Workloads: The ability to generalize from benchmark workloads to real-user workloads is crucial to the adaptation of our models to real-world deployments. To evaluate generalizability of our models, in PARIS, we trained our models using commonly deployed cloud-hosted applications, such as video encoding, compression and serving-style workloads running on common cloud serving datastores such as Aerospike, MongoDB, Redis, and Cassandra. More details on the benchmark workloads suite used and datasets collected are provided in Chapter 5.

1.4 Organization

This dissertation incorporates our previously published work [160, 92, 161, 162] and is organized as follows.

Chapter 2 describes relevant details of current data analytics frameworks and provide background on relevant machine learning terminologies used in this dissertation.

Chapter 3 describes Wrangler [160], a predictive scheduler that allows data intensive computational jobs to be predictable and faster while using fewer resources compared to existing reactive mechanisms. We describe how we deal with the challenge of model uncertainty and achieve significant improvements in job completion times.

Chapter 4 describes our multi-task learning formulations that share information between the various models that are *similar*, allowing us to use less training data and bring training time down significantly. We then extend these formulations using group sparsity inducing norms to automatically discover the similarities between models and improve interpretability.

Chapter 5 describes PARIS [162], a data-driven system address the fundamental problem of accurately and economically choosing the best VM for a given workload and user goals.

Chapter 6 surveys existing resource management mechanisms.

Chapter 7 concludes with some future directions for research.

Chapter 2

Background

In this chapter we describe the architecture of parallel data analytics frameworks and provide background on the ML terminologies used in this dissertation.

2.1 Architecture of computational frameworks

Analytics on huge datasets available today has become a driving factor for businesses. Parallel processing on commodity clusters has emerged as the de-facto way of dealing with the scale and complexity of such data-intensive applications. To accelerate applications using multiple machines, a bulk-synchronous-parallel model of distributed computation is used. In a bulk-synchronous-parallel model, a computational job submitted to a cluster of computers is first broken into stages and then each stage is split into multiple tasks. Tasks in a stage are assigned the same computation but different blocks of input data. Tasks belonging to each stage can run in parallel. A cluster scheduler assigns these tasks to machines (nodes), where they are executed in parallel. A job's computation synchronizes along the boundaries of different stages: output of previous stage is used as input by the next stage. This shuffle of intermediate data is generally executed by tasks from a previous stage writing the data to the local disk and tasks from the next stage reading it. A job finishes when all its tasks have finished execution.

A key feature of such frameworks is that they automatically handle failures (which are more likely to occur on a cluster of failure-prone commodity computers) without needing extra efforts from the programmer. If a node crashes, a parallel data-analytics framework re-runs all the tasks it was executing on a different node.

Google originally proposed the MapReduce [56] framework that divides each job in exactly two stages: a map stage and a reduce stage. MapReduce is highly scalable to large clusters of inexpensive commodity computers. Hadoop [155], an open source implementation of MapReduce, has been widely adopted by industry over the last decade. Later, Spark [171], Dryad [90], and Naiad [113] were designed to support more ad-hoc parallel processing computational jobs that are represented by a more general DAG (directed acyclic graph) of one of more stages.

2.2 Job scheduling in computational frameworks

We now describe job scheduling in parallel data intensive computational frameworks. In the interest of simplicity of explanation, we assume a cluster with single master and multiple worker machines. The results and discussion in this dissertation apply to other more sophisticated architectures such as clusters with multiple master and multiple worker nodes.

In the most simple form, job scheduling mechanism in parallel data analytics frameworks is orchestrated by a central scheduler. Jobs are submitted to this central scheduler that divides each job in stages and represents them with a directed acyclic graph of computations. Each stage is divided into multiple tasks. The central scheduler then assigns these tasks to worker machines. This assignment of tasks to worker nodes is done based on availability of 'slots' on them. More complex mechanisms use fairness, bin-packing for scheduling that aim for improved performance and resource usage of tasks (see Chapter 6 for a survey on cluster schedulers).

2.3 Relevant machine learning preliminaries

In this section, we present background on relevant machine learning terminologies used in this dissertation.

2.3.1 Supervised learning

Supervised learning is a branch of machine learning algorithms where predictive models are built from known input and target data pairs, called the *labeled dataset*. Generally, a given labeled dataset is split into *training set, validation set, and a test set*. Supervised learning algorithms aim at learning a function, also called a model, that can map the input data to the corresponding target values in the training dataset. The learned models are validated on the validation set that the model has not seen during its training phase. A model that performs the best according a metric of success, such as prediction accuracy, on the validation set is selected as an output of the *training phase*. In the *test phase*, the selected model is then used to predict the target values, called *labels*, of input data in the test set. The labels in the test set, called *ground truth*, are used to evaluate the performance of the learned model.

Supervised learning includes two categories of algorithms: (a) Classification: for categorical label values, and (b) Regression: for real-valued labels.

More formally, the supervised learning setup used in this dissertation can be presented as follows: Let D be a labeled dataset with n datapoints,

$$D = \{(x_i, y_i) : 1 \leq i \leq n\}$$

where, $x_i \in R^d$, and $y_i \in \{0, 1\}$ for *binary classification*. In the regression setting, $y_i \in R$.

Let D_{train}, D_{val}, and D_{test} be the train, validation and test splits of dataset D.

$$D_{train} = \{(x_i^{train}, y_i^{train}) : 1 \leq i \leq n_{train}\}.$$

$$D_{val} = \{(x_i^{val}, y_i^{val} : 1 \leq i \leq n_{val}\}.$$
$$D_{test} = \{(x_i^{test}, y_i^{test} : 1 \leq i \leq n_{test}\}.$$

where, $n = n_{train} + n_{val} + n_{test}$.

The goal is to learn a function, $f : x_i \longrightarrow y_i, \forall(x_i, y_i) \in D_{train}$, using the validation set, D_{val}, and evaluate it on the test set, D_{test}.

2.3.2 Support Vector Machines

Support vector machines (SVMs) have become a popular off-the-shelf supervised learning algorithm over last couple of decades, especially due to its better empirical performance compared to other algorithms. SVM uses machine learning theory to maximize predictive accuracy while automatically avoiding over-fit to the training data.

Vapnik [148] developed the foundations of SVM using the Structural Risk Minimization (SRM) or the regularized Empirical Risk Minimization (ERM) principle [36] that allows SVM to generalize well to unseen datasets. To find the decision function that can predict the labels for an unseen dataset, an SVM solver minimizes a general objective provided by the framework of regularized empirical risk minimization.

Loss functions. A loss function measures the quality of a model's prediction. More concretely, it measures how far away the predicted output is from the true label. For a given datapoint, (x, y), let the model's prediction be $f(x)$. The loss can be denoted by $L(x, y, f(x))$. Examples of commonly used loss functions include, L_1 loss, L_2 loss (see [26] for more details).

Empirical risk. To build a model that is capable of making accurate predictions on a test dataset, we must minimize the training set error of our predictions. The training dataset is assumed to be drawn from the same distribution.

The empirical risk of a given prediction function is defined as the expected value of the loss of our function on a given dataset. We can write empirical risk, $\hat{R}(f)$ as,

$$\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n} L(x_i, y_i, f(x_i)).$$

To build an accurate model, we minimize the empirical risk.

***Regularized* empirical risk minimization.** SVM solvers minimize *regularized* empirical risk or structural risk that avoids overfitting and enables generalizable models that produce accurate predictions for previously unseen data.

SVM solvers typically find a decision function represented by a hyperplane, $\{x \in R^d : f(x) = w^T x - b\}$ by minimizing regularized ERM over $(\mathbf{w}, b)$.

The regularized ERM specifies a bi-objective:

$$\hat{R}(f) + \gamma(w).$$

Regularized ERM is the sum of empirical risk, $\hat{R}(f)$ and a regularizer, $\gamma(w)$. See a tutorial on Support Vector Machines [36] by Burges, et al., for further details.

Chapter 3

Predictive Scheduling: Predictable and Faster Jobs using Fewer Resources

In the previous chapter, we described the architecture of today's analytics frameworks. Reducing the job completion time for data intensive applications running atop such distributed processing frameworks [56, 90] has attracted significant attention recently [14, 170, 12, 11]. In this chapter, we focus on achieving predictable and faster job completion times while using fewer resources. Slow running tasks, called *stragglers*, impede job completions. Prior works assume stragglers are hard to predict in advance. We show that performance and utilization indicators can be leveraged to predict stragglers. We describe how we posed straggler prediction as an instance of binary classification where we classify the state of underlying machines as healthy or straggler-prone. We then feed these predictions to the scheduler as hints to build a predictive scheduler, *Wrangler*, that proactively avoids straggler-causing situations. Prediction errors of data-driven models pose a key challenge to their robustness. We demonstrate how we deal with this challenge to enable significant gains in job completions times.

3.1 Introduction

A major challenge to achieving near-ideal job completion time is the slow running or *straggler* tasks – a recent study [11] shows that despite existing mitigation techniques, straggler tasks can be 6-8$\times$ slower than the median task in a job on a production cluster. This leads to high job completion times, over-utilization of resources and increased user costs. Mitigating or even eliminating stragglers thus remains an important problem.

Existing approaches aren't enough to solve this problem. *Speculative execution* [56] is a replication-based reactive straggler mitigation technique that spawns redundant copies of the slow-running tasks, hoping a copy will reach completion before the original. This is the most prominently used technique today, including production clusters at Facebook and Microsoft Bing [12]. However, without any additional information, such reactive techniques cannot differentiate between nodes that are inherently slow and nodes that are temporarily overloaded [55]. In the latter case, such techniques lead to unnecessary over-utilization of resources without necessarily improving the job completion times. Though proactive, Dolly [11] is still a replication-based approach that focusses only on interactive jobs and incurs extra resources. Being agnostic to the correlations between stragglers and nodes' status, replication-based approaches are wasteful.

Another proactive alternative is to model running tasks statistically to predict stragglers. However, realistic modeling of tasks in cluster environments is difficult due to complex and unpredictable interactions of various modules [12, 54, 15, 69, 47, 11, 55]. Black box approaches can learn these interactions automatically [32, 79]; however, these techniques are opaque and prone to errors that could lead to inefficient utilization and longer completion times [11].

To avoid such problems, a straggler mitigation approach should meet the following requirements:

- It should not wait until the tasks are already straggling.
- It should not waste resources for mitigating stragglers.

To this end, we introduce *Wrangler*, a system that predicts stragglers using an interpretable linear modeling technique based on cluster resource usage counters and uses these predictions to inform scheduling decisions. This allows it to avoid waiting until tasks are already running slow. *Wrangler* prevents wastage of resources by removing the need for replicating tasks. Furthermore, *Wrangler* introduces a notion of *confidence measure* with these predictions to overcome the modeling error problems; this confidence measure is then exploited to achieve a *reliable* task scheduling. In particular, by using these predictions to balance delay in task scheduling against the potential for idling of resources, *Wrangler* achieves a speed up in the overall job completion time. A threshold on confidence measures can be tuned to suit different workloads to avoid costly incorrect scheduling decisions. A prototype implementation of *Wrangler* demonstrates up to 61% improvement in overall job completion times while reducing the resource consumption by up to 55% for production-level workloads using a 50 node EC2 cluster.

3.2 Prior Work and Motivation

Existing approaches to straggler mitigation fall short in multiple ways: reactive approaches act after tasks have already slowed down. Replication-based approaches, whether proactive or reactive, use extra resources. White or gray box approaches depend on causes that keep changing dynamically and hence are difficult to enumerate *a priori*. Finally, black box approaches are prone to modeling errors. We discuss each of these in detail below.

Reactive techniques rely on a *wait-and-speculate* re-execute mechanism. Speculative execution, a widely used approach for straggler mitigation, marks slow running tasks as stragglers and reacts by relaunching multiple copies of them. This is inefficient because a task is allowed to run for a significant amount of time before it can be identified as a straggler. By this time, other tasks of that job have made considerable progress already, increasing the possibility of extending the job's finishing time. SkewTune [98] avoids replicating tasks but is still a wait-and-speculate mechanism.

Replication-based approaches incur extra resources. For instance, speculative execution launches multiple redundant copies of the same task. As soon as one of these copies finishes execution, the rest are killed. This amounts to wastage of resources. LATE [170] improves over speculative execution using a notion of progress scores, but still results in resource waste. Cloning mechanisms [11], being replication-based, also incur extra resources.

Though there are no existing mechanisms that proactively avoid stragglers without replication, scheduling or load-balancing approaches [166, 168, 80, 3, 91, 13] indirectly attempt to do so by reducing resource contention on a cluster. The delay scheduler [168], for example, focuses on reducing network bottlenecks. However, we show in Section 3.4.2.2 that despite these efforts, stragglers still occur. Being focused on load balancing in heterogeneous clusters, Tarazu [3] ignores temporary overloading that occurs even in homogeneous clusters.

White or gray box approaches, whether reactive [14] or scheduling-based [80, 166, 168], attempt to identify stragglers based on whether local conditions exceed fixed thresholds defined over a set of resource usage statistics. However, in many cases, given resource usage statistic values sometimes result in a straggler, and other times do not. Further, in Section 3.4.2.2, we illustrate how *the contributors of straggler behavior vary across nodes and over time* in the Facebook and Cloudera real-world production traces. For example, we observed that two disk-intensive tasks scheduled on a node with slow disk were found to be straggling. The same tasks co-located on another node, however executed normally, as the other node had enough disk bandwidth. But when we avoided scheduling such tasks simultaneously on the node with slow disk, the tasks straggled due to other resource contention patterns, including network and memory. Even if we could easily distinguish conditions that always result in stragglers, it is infeasible to exhaust the space of all possible resource usage statistic settings to classify them as such. This limits the usefulness of white or gray box approaches.

An alternative is to use black box approaches to automatically learn a node's behavior. Bortnikov, et al., [32] have shown that high-quality predictions of stragglers can be made, although they did not attempt to incorporate such predictions into a scheduler. Incorporating such techniques into a scheduler could backfire during real-life deployment on production clusters [11]. To make these black box techniques more transparent, it is critical to be able to assess what the technique is learning from the data. Further, modeling errors might render the system's performance unstable. It is crucial to be robust against such errors for achieving predictable performance. Since off-the-shelf learning techniques do not include mechanisms for evaluating the quality of their output, these are not sufficient to tackle this problem. Despite these challenges, there is evidence to believe that machine learning techniques can be successfully incorporated into production systems: previous works [79, 58] have demonstrated the use of machine learning for selecting what resources to assign to a task. However, they did not perform straggler mitigation.

To address each of the issues identified above, Wrangler provides:

1. Interpretable models that can predict what conditions will lead to stragglers using readily available performance counters. These models automatically adapt to the dynamically changing resource usage patterns across nodes and across time. This capability of predicting if a node could cause a straggler opens up various avenues for avoiding stragglers.

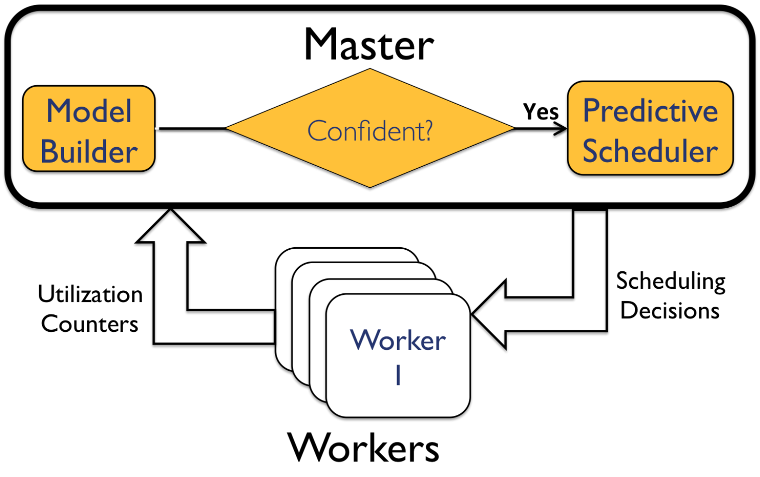


Figure 3.1: Architecture of *Wrangler*.

2. Confidence measures that guard against modeling errors by telling us when the models are sure of their predictions. Modeling errors fundamentally limit real life applicability of previously proposed approaches [32, 79]. *Wrangler* addresses this by providing configurable confidence measures. We show in Section 3.7.4 that the use of confidence measures is crucial.

3. A task scheduling mechanism that incorporates these predictions to improve overall job finishing times. This reduces resource consumption compared to replication-based mechanisms.

Although it serves as a straggler avoidance approach on its own, *Wrangler* can also be used in conjunction with existing mitigation approaches.

3.3 Our Proposal: *Wrangler*

Given resource utilization metrics of a node, *Wrangler* predicts if a newly assigned task on that node will turn out to be a straggler. It then uses these predictions to make scheduling decisions that proactively avoid straggler formation.

3.3.1 Architecture of *Wrangler*

Figure 3.1 shows *Wrangler's* system architecture that extends the architecture of Hadoop. Job scheduling in Hadoop is handled by a master that controls the workers. The master assigns tasks to worker nodes in response to the heartbeat message sent by them every few seconds. The assignments depend upon the number of available slots as well as locality. *Wrangler* has two basic components.

1. **Model Builder:** Using the job logs and snapshot of resource usage counters collected regularly from the worker nodes using a Ganglia [107]-based node-monitor, we build a model per node.

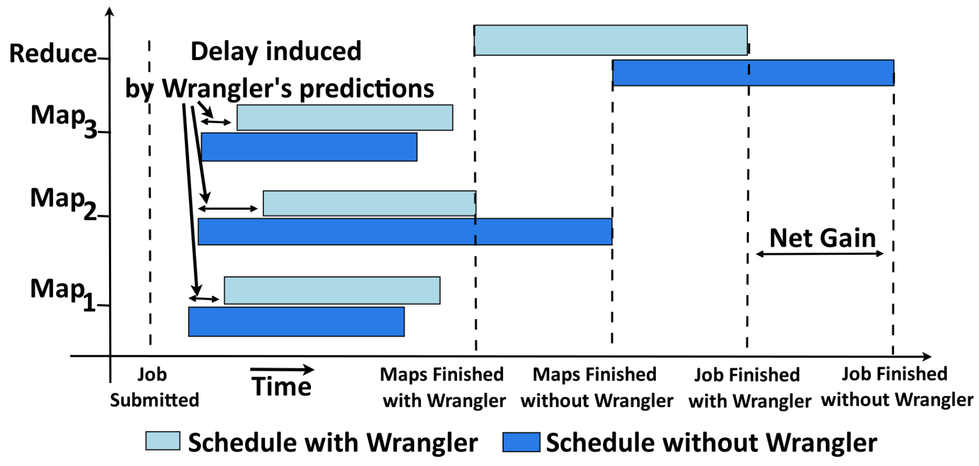

Reduce
Delay induced by Wrangler's predictions
Map3
Map2
Map1
Net Gain
Job Submitted
Time
Maps Finished with Wrangler
Maps Finished without Wrangler
Job Finished with Wrangler
Job Finished without Wrangler
Schedule with Wrangler
Schedule without Wrangler

Figure 3.2: Lifetime of an example job observed with and without *Wrangler*. Note: there are other jobs (not shown here) executing simultaneously on the cluster. The careful assignment of Map_2 to a node, that is likely to finish it in a timely manner, accelerates the job completion despite the delay introduced in its launching.

These models predict if a task will straggle given its execution environment; they also attach confidence measures to their predictions. Section 3.4 describes the Model builder in detail.

2. **Model-informed Scheduler:** Using the predictions from the models built earlier, a model-informed scheduler then selectively delays the start of task execution if that node is predicted to create a straggler. A task is delayed only if the confidence in the corresponding prediction exceeds the minimum required confidence. This avoids overloading of nodes, thus reducing their chances of creating stragglers. Section 3.5 details the Model-informed scheduler.

Our main tool is to defer a task's assignment until a node, that is likely to finish it in a timely manner, is found. Figure 3.2 shows an example job with three mappers and a reducer. Without *Wrangler*, Map_2 was a straggler, as its *normalized duration*, the ratio of its duration to the size of its input data, was much larger than the other mappers. When *Wrangler* predicted it to be a straggler on this node, this assignment was avoided. This decision introduced a delay in the task's start until the same node is no longer overcommitted or a different non-overcommitted node is found. Due to this assignment, Map_2 finished faster than it did without *Wrangler*. The reducer then started earlier and we achieved a net improvement in the job duration.

3.3.2 Novelty of our Approach

Wrangler takes a radically different approach compared to previous straggler mitigation strategies by predicting straggler tasks before they are even launched and scheduling them well to avoid their occurrence in the first place. *Wrangler* achieves its goal by collecting extensive information and deriving useful correlations automatically. According to our observations, what causes stragglers varies across nodes and time. Being a learning-based approach, *Wrangler* is capable of adapting to various situations that cause stragglers. It can figure out for itself what factors are causing tasks to run slower than usual. Importantly, the straggler prediction models we build are interpretable; meaning that we can gain insights from what these models learn using the data (Section 3.4.2.2). This relieves us from having to explicitly diagnose each case manually, which as we argued earlier is infeasible. Note that even in presence of more sophisticated schedulers, *Wrangler's* ability of adapting to dynamically changing cluster execution environments and changing resource patterns justifies its applicability. Additionally, the role of *confidence measure* is crucial for handling modeling errors. This allows our probabilistic learning-based approach to be robust. Learning techniques do not compute confidence measure. Our work is the first to introduce the use of confidence measures to ensure stability of straggler prediction models. Further, *Wrangler*'s model-informed scheduler induces delays in launching tasks on nodes that are predicted to create stragglers. In the worst case, due to possible prediction errors, our approach could lead to bounded suboptimal performance. Our approach however, does not lead to incorrect execution, termination or replacement of tasks; thus maintaining liveness and correctness guarantees.

3.4 Building Straggler Prediction Models

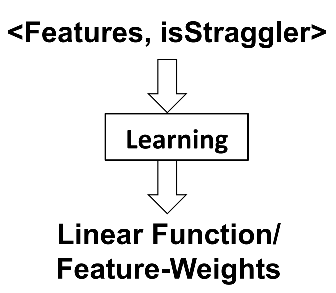
The aim of *Wrangler's model builder* component is to build *accurate* straggler prediction models such that they are *robust* with respect to possible modeling errors and are *interpretable* with respect to what they learn.

3.4.1 Linear Modeling for Predicting Stragglers

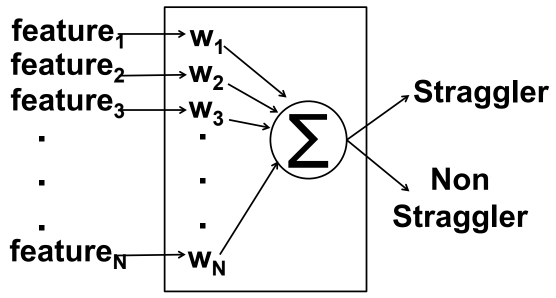
As we mentioned earlier, finding what actually causes stragglers is challenging due to complex task-to-node and task-to-task interactions. To capture these complex interactions, we collected numerous resource usage counters from the cluster using Ganglia [107]. Linear modeling techniques [52] from the machine learning domain are appropriate for probabilistic modeling of a node's behavior, which can be represented through the various resource usage counters. These techniques adapt to dynamically changing resource usage patterns on a node. This alleviates the pains of manual diagnosis of the source of individual straggler appearance. We learn the behavior of each node individually to be robust to heterogeneity in today's clusters.

As shown in Figure 3.3a, during the learning phase, these techniques learn *weights* on the features using labeled data that represents the ground truth. In our context this data is the node's resource usage counters at the time of submission of a task and a label (isStraggler), indicating whether it was a straggler. Using these weights and the node's resource usage counters the model calculates a score for predicting if it will turn out to be a straggler. This prediction phase is depicted in Figure 3.3b. Next, we provide a high-level intuitive understanding of one such linear modeling technique that we use, Support Vector Machines (SVM) with linear kernels. For mathematical details, see [26, 36].[1]

[1]Other interpretable classification techniques, such as decision trees could also be used, as used in [32]. We found their performance to be similar.



(a) Learning phase



(b) Prediction phase

Figure 3.3: Linear modeling techniques for predicting stragglers: node's resource usage counters form the features. The Learning phase (a) learns the weights on features using the ground truth, i.e., a labeled dataset that consists of (1) features of a task and (2) whether it was an straggler or not. A linear model then predicts based on a score obtained by taking a linear combination of features with corresponding weights (b).

3.4.1.1 Support Vector Machines for Predicting Stragglers

SVM is a statistical tool that learns a linear function separating a given set of vectors (e.g., node's resource usage counters) into two classes (e.g., straggler class and non-straggler class). This linear function is called the *separating hyperplane*; each of the two half spaces defined by this hyperplane represents a class. In the model building phase, this hyperplane is computed such that it separates the vectors of node's resource usage counters belonging to one class (stragglers) from those of the other class (non-stragglers) with maximum distance (called margin) between them. Later, a new observed resource usage vector (i.e., a test vector) can be evaluated to see which side of the separating hyperplane it lies, along with a score to quantify the confidence in classification based on the distance from the hyperplane.

3.4.1.2 Features and labels

To predict whether scheduling a task at a particular node will lead to straggler behavior, we use the resource usage counters at the node. The MapReduce paper [56] mentioned that stragglers could arise due to various reasons such as competition for CPU, memory, local disk, network bandwidth. LATE [170] further suggests that stragglers could be caused due to faulty hardware and misconfiguration. Mantri [14] reports that the dynamically changing resource contention patterns on an underlying node could give rise to stragglers. Based on these findings, we collected the performance counters for CPU, memory, disk, network, and other operating system level counters describing the degree of concurrency before launching a task on a node. The counters we collected span multiple broad categories as follows:

1. *CPU utilization:* CPU idle time, system and user time and speed of the CPU, etc.

2. *Network utilization:* Number of bytes sent and received, statistics of remote read and write, statistics of RPCs, etc.

3. *Disk utilization:* The local read and write statistics from the datanodes, amount of free space, etc.

4. *Memory utilization:* Amount of virtual, physical memory available, amount of buffer space, cache space, shared memory space available, etc.

5. *System-level features:* Number of threads in different states (waiting, running, terminated, blocked, etc.), memory statistics at the system level.

In total, we collect 107 distinct features characterizing the state of the machine. We refer the reader to Ganglia [107, 68] for detailed explanation of the performance metrics. See Section 3.4.2 for other details of our dataset.

Features: Multiple tasks from jobs of each workload may be run on each node. Therefore to simplify notation, we index the execution of a particular task by i and define $S_{n,l}$ as the set of tasks corresponding to workload l executed on node n. *Before* executing task $i \in S_{n,l}$ corresponding to workload l on node n we collect the resource usage counters described above on node n to form the feature vector $x_i \in \mathbb{R}^{107}$. For each feature described above we subtract the minimum across the entire dataset and rescale so that it lies between 0 and 1 for the entire dataset.

Labels: After running task i we measure the normalized task duration $nd(i)$ which is the ratio of task execution time to the amount of work done (bytes read/written) by task i. From the normalized duration, we determine whether a task has *straggled* using the following definition:

Definition A task i of a job J is called a *straggler* if

$$nd(i) > \beta \times \underset{\forall j \in J}{\text{median}}\{nd(j)\} \tag{3.1}$$

where $nd(i)$ is the normalized duration of task i computed as the ratio of task execution time to the amount of work done (bytes read/written) by task i.

We use a value of 1.3 for β in most of our experiments. In Section 3.4.2.2, we show that our models are agnostic to the value of β. Given the definition of a straggler, for task i we define $y_i \in \{0, 1\}$ as a binary label indicating whether the corresponding task i ended up being a straggler relative to other tasks in the same job.

3.4.1.3 Prediction Task

While the resource usage counters do track the time-varying state of the node, they do not model the variability across nodes, or the properties of the particular task we are executing. To deal with the variability of nodes, Wrangler builds a separate predictor for each node.

Modeling the variability across tasks is hard since it would require understanding the code that the task is executing. Instead, Wrangler uses the notion of "workloads", which we define next. Companies such as Facebook, Google, use compute clusters for various computational purposes. We call the specific pattern of execution of jobs on these clusters a *workload*. These workloads are specified using various statistics, such as submission times of multiple jobs, number of their constituent tasks along with their input data sizes, shuffle sizes, and output sizes. In this work, we assume that all tasks in a particular workload have similar properties in terms of resource requirements, and capture the variability across workloads by building separate predictors for each workload. Thus Wrangler builds separate predictors for each node and for every workload.

Putting it all together, we can state the binary classification problem for predicting straggler tasks more formally as follows. A datapoint in our setting corresponds to a task i of job J from a particular workload l that is executed on a node n in our cluster. Before running task i we collect the features x_i which characterize the state of node n. After running task i we then measure the normalized duration and determine whether the task straggled with respect to other tasks of job J (see Definition 3.1). Our goal is to learn a function:

$$f_{n,l} : x \longrightarrow y,$$

for each node n and workload l that maps the current characteristics $x \in \mathbb{R}^d$ of that node (e.g., current CPU utilization) to the binary variable $y \in \{0, 1\}$ indicating whether that task will straggle (i.e., run longer than β times the median runtime).

3.4.1.4 Training

For any workload l, Wrangler first collects data and ground truth labels for training and validation. In particular, for every task i launched on node n, Wrangler records both the resource usage counters x_i and the label y_i which indicates if the task straggles or not. Since there is a separate predictor for each node and workload, Wrangler produces separate datasets for each node and workload. Let $S_{n,l}$ be the set of tasks of jobs corresponding to workload l, executed on node n. Thus, we record the dataset for node n, workload l as:

$$D_{n,l} = \{(x_i, y_i) : i \in S_{n,l}\}.$$

Then Wrangler divides each dataset into a training set and test set temporally, i.e, tasks belonging to the first few jobs constitute the train set and the rest form the validation. The predictors are then trained on these datasets.

3.4.1.5 Imbalance in the dataset

Various modeling techniques are sensitive to imbalanced datasets. Non-straggler tasks outnumber the stragglers causing an imbalance in the dataset used for building models. Due to the way underlying optimization problems are formulated, the predictions favor the class with a majority of instances. In this context, every task is predicted to be a non-straggler. Ideally, the best results are obtained when each class is represented equally in the learning dataset. We statistically oversample [41, 40, 141, 174] the instances from the minority class (i.e. straggler class) which is a common technique for dealing with imbalanced datasets.

3.4.1.6 Test Phase

After this initial training phase, the classifiers are frozen, and the trained classifiers are incorporated into the job scheduler as described above. They are then tested on the next $T_{test} = 10$ hours by measuring the impact of these classifiers on job completion times.

3.4.1.7 Confidence Measure

Simply predicting a task to be a 'straggler' or a 'non-straggler' is not robust to modeling errors. To ensure reliable predictions, we introduce the notion of *confidence measure* along with the prediction of these linear models. We need a confidence measure to help decide if our predictions are accurate enough for preventing stragglers by influencing the scheduling decisions. Wrangler uses SVMs with scaling by Platt [123] to produce a confidence: we next explain this computation intuitively and then mathematically. The farther a node-counter vector is from the separating hyperplane, higher are the chances of it belonging to the the predicted class. To obtain a probability estimate of the prediction being correct, we can convert the distance from the separating hyperplane to a number in the range [0, 1]. We obtain these probabilities by fitting logistic regression models to this distance [123]. Next, we explain how to compute a confidence measure.

A snapshot of a node's resource usage counters represents the node at a given time instant. We denote this set of features as a vector x. The SVM outputs a linear hyperplane of the form $w^T x + b$, where w is a vector of weights learned by SVM corresponding to the features. Data points which have a positive score, i.e., $w^T x + b > 0$ are classified as stragglers and points which have a negative score are classified as non-stragglers. The SVM doesn't itself output probabilities, but the score ($w^T x + b$), which is also the distance of a point to the hyperplane, serves as a measure of the classifier's confidence. Points far away from the hyperplane (i.e., with high positive scores or highly negative scores) are those which the classifier is very confident about. We train a logistic regression classifier to convert this score into probabilities. The logistic regressor outputs a probability of the form $\frac{1}{1+exp(-\alpha s-\beta)}$ where s is the score and α and β are parameters that are estimated using logistic regression [26].

3.4.1.8 Interpretability

Our straggler prediction models are interpretable in that they allow us to gain insights from what they learned using the data. These models automatically learn the contribution of a feature towards creation of a straggler, called *weight* of that feature. We bring out the causes behind stragglers using these weights assigned to the features.

For a given task, our models (SVM with linear kernel [26, 36]) predict based on a linear combination of features with their corresponding weights. In other words, their predictions are based on a score obtained by multiplying feature-values with their respective weights and adding these products. We then analyze the resource usage counters of actual straggler tasks. Given the node's resource usage counters at the launch time of such an actually observed straggler task, we want to find out the set of features that primarily caused it to be a straggler using the weights learned by our models. Since it is tedious to capture a holistic picture with over 100 feasible node resource utilization counters involved, we grouped them into five feature categories; CPU utilization, network utilization, disk utilization, memory utilization and other system-level features. We then selected a subset of these features that makes up at least 75% of the models' score for the given task. Since this subset of features has driven the model's decision to predict it to be a straggler, we deem this as the *cause* behind this straggler. We then find out the fraction of straggler tasks on that node that have the same cause. In Section 3.4.2.2, we present this analysis.

Next, we evaluate prediction accuracy of SVM on production traces from Facebook and Cloudera. Section 3.5 describes how we used these predictions with associated confidence for affecting scheduling decisions to avoid stragglers.

3.4.2 Model-Builder Evaluation

We evaluate the straggler prediction models on real world production-level workloads from Facebook and Cloudera by replaying them on a 50 node EC2 cluster as explained below. A prediction is correct if it matches with the actual label. We evaluate the models based on (1) how many times they predicted straggler tasks correctly: percentage true positives and (2) how many times they mis-predicted a non-straggler task to be a straggler: percentage false positives. High true positive and low false positive values indicate a good quality model.

3.4.2.1 Experimental set-up

Production-level Workloads: We learn to predict stragglers based on production level workload traces from multiple Hadoop deployments including those at Facebook and Cloudera's customers. Our dataset covers a wide set of workloads allowing for a better evaluation. Table 3.1 provides details about these workloads in terms of the number of machines, the length and date of data capture, total number of jobs in those workloads. Chen, et al., explain the data in further details in [43]. Together, the dataset consists of traces from over about 4600 machines captured over almost a year.

For faithfully replaying these real-world production traces on our 50 node EC2 cluster, we used a statistical workload replay tool, SWIM [44] that synthesizes a workload with representative job submission rate and patterns, shuffle/input data size and output/shuffle data ratios (see [44] for details of replay methodology). SWIM scales the workload to the number of nodes in the experimental cluster. As our testbed, we used a cluster of 50 nodes on Amazon EC2. SWIM submits the jobs and its constituent tasks to this cluster according to the workload it synthesized based on the production trace. Next we describe how we collect our dataset capturing the resources utilized on the nodes in our cluster when SWIM replays these workloads.

Dataset: For building straggler prediction models, we need a labeled dataset that consists of a number of {features, label} pairs. In this context, features are the resource usage counters of a node at the time of submission of a task; and label is whether it was a straggler or not. To generate this dataset, we replayed the production-level traces (see Table 3.1), using SWIM [44] on Amazon

Trace	#Machines	Length	Date	#Jobs
FB2009	600	6 month	2009	11,29,193
FB2010	3,000	1.5 months	2010	11,69,184
CC_a	100	1 month	2011	5,759
CC_b	300	9 days	2011	22,974
CC_d	400-500	2+ months	2011	13,283
CC_e	100	9 days	2011	10,790
Total	≈ 4,600	≈ 11.5 months	-	23,51,183

Table 3.1: Dataset. *FB*: Facebook, *CC*: Cloudera Customer.

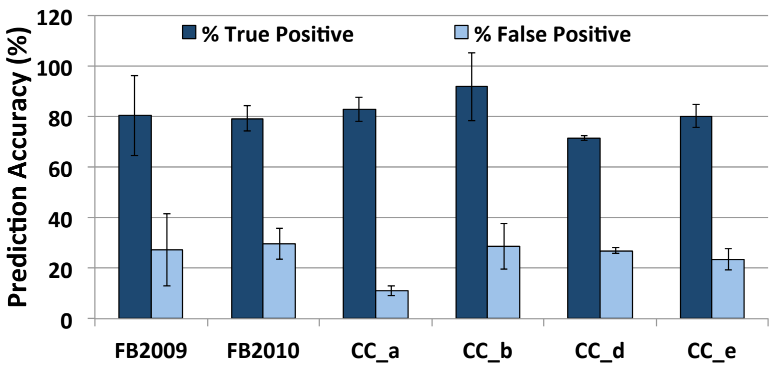


Figure 3.4: Classification accuracy of SVM for all workloads.

EC2 cluster of 50 m1.xlarge instances. Using Ganglia [107] we captured the node's resource usage counters at regular intervals of 15 seconds; this forms the features in the dataset (see Section 3.4.1.2). We label the dataset by marking straggler tasks based on definition 3.1. In Section 3.4.2.2, we show that our models are agnostic to the value of β used in defining stragglers.

3.4.2.2 Results of Model-evaluation

Figure 3.4 shows the straggler prediction accuracy on all the workloads using SVM. We use the tasks executed over initial 2 hours to build models and test their accuracy on the tasks submitted over the next ten hours. Overall, using a linear kernel SVM with Sequential Minimal Optimization [124], we obtain about 80% true positive and about 30% false positive percentages. This means that we predicted 80% of total stragglers tasks as stragglers and mis-predicted 30% of non-stragglers to be stragglers. This completes the model building phase and then we deploy the models so that they provide hints to *Wrangler's* scheduler. In Section 3.7, for example, we show that we achieve 61% and 43% improvement in 99^{th} percentile of overall completion times for *FB2009* and *CC_b* workloads respectively. This confirms that about 80% true positive percentage is good enough.

Sensitivity of the models to the definition of stragglers: Based on the value chosen for β, number of stragglers vary (see Definition 3.1). Intuitively, β indicates the extent to which a task is

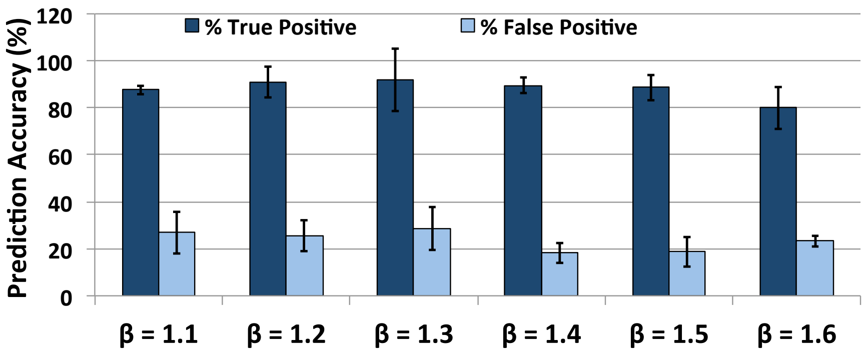


Figure 3.5: Sensitivity of classification using SVM with respect to values of β for workload CC_b. The error bars show the standard deviation across the percentage accuracy of models built for the 50 nodes in the cluster. Note: The valid range for β shown is $\beta > 1$ ($\beta = 1$ is simply the median) to $\beta < 1.7$ (lower enough to ensure minimum number of stragglers for oversampling and training).

allowed to slow down before it is called a straggler. Our mechanism is agnostic to the value of β chosen for all the workloads. We show a representative sensitivity analysis with respect to β for the CC_b workload in Figure 3.5. We chose β from the range $(1, 1.7)$. $\beta = 1$ is simply the median and we did not see enough stragglers for β greater than 1.6 for CC_b to be able to oversample and build a model. In our experiments, we set β = 1.3.

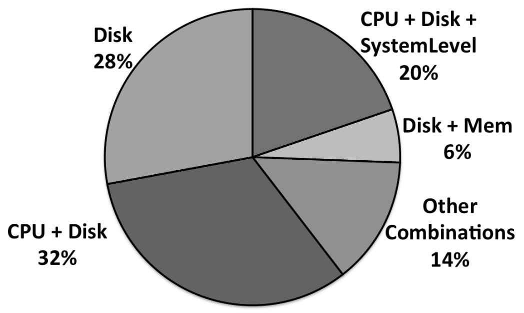
Insights Provided by the Models: We briefly describe the insights we obtained from the straggler prediction models about the causes behind them. However, we leave the detailed explanation of the causes for future work. In Section 3.7, we show that *Wrangler's* model-informed scheduler accelerates job completion by making careful task-to-node assignments; this avoids such straggler-causing situations.

Figure 3.6 presents the percentage of stragglers created due to different causes. Figure 3.6a shows that for *FB2010*, disk utilization (I/O) was the primary bottleneck creating temporary hotspots along with interference with simultaneously executing tasks' memory, CPU usage patterns. Figure 3.6b shows the causes behind stragglers on another node in the cluster executing the same workload (*FB2010*). Although the disk (I/O) usage still dominates, other features also contribute considerably to the creation of stragglers on this node. On a node executing CC_b, as shown in Figure 3.6c, memory contention contributed the most in creating stragglers.

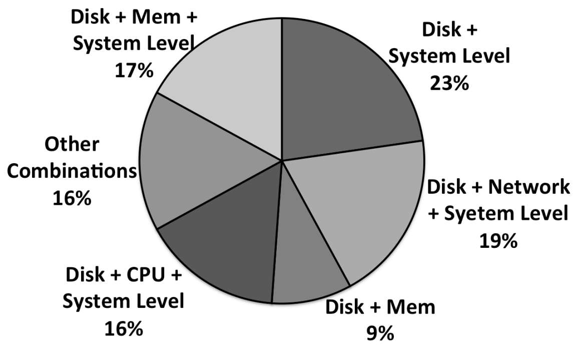
From our analysis of causes behind stragglers indicated by the models, we note the following:

- Causes behind stragglers vary across nodes – this is true even for the clusters of the same instance types on Amazon EC2. This justifies our approach of building a straggler prediction model per node. This decision also makes our approach robust to heterogeneity.

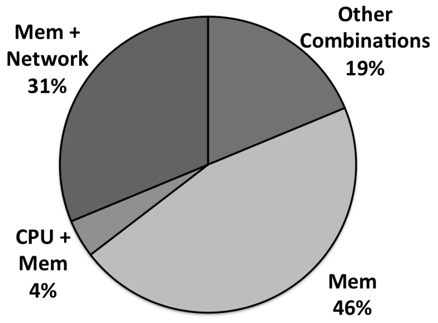
- Causes vary across workloads – we see that for *FB2010*, dominating contributor was disk usage whereas the tasks of the CC_b workloads contend over memory. In [43], Chen et al., explain that *FB2010* is I/O intensive, supporting this insight obtained from our models.



(a) FB2010, $Node_1$



(b) FB2010, $Node_2$



(c) CC_b, $Node_1$

Figure 3.6: Causes behind stragglers

- Network utilization features were not seen to be the prime contributors in our experiments for any of the workloads we evaluated on. We believe network utilization was not the bottleneck as we had enabled the locality-aware delay scheduling mechanism [168]. This indicates that since none of the existing schedulers are straggler-aware, they cannot eliminate stragglers.
- Complex task-to-task interactions on an underlying node tend to create temporary hotspots. Lack of this information can cause scheduling decisions to go wrong.

Note that it is hard to know the contributors to the straggling behavior of tasks *a priori* without the help from the models. This justifies our use of learning-based models that automatically adapt to various causes behind stragglers. Using the straggler prediction models, we can proactively inform the schedulers of such straggler causing situations. In Section 3.5, we propose such a scheduler that exploits these predictions to avoid creating stragglers in the first place.

3.5 Model-informed Scheduling

In this section, we describe our scheduling algorithm that uses the straggler predictions to selectively delay task assignment to nodes. We then explain the significance of tuning parameters of this algorithm and describe how we learn their values. Finally, we conclude with a theoretical analysis that bounds the delays our algorithm can introduce.

Algorithm 1 Model-informed scheduling algorithm

```
Let $N$={$n_i$ : $i$=1, ..., # workers} be the set of worker nodes
Let $willStraggle_i \in$ {yes, no} be the prediction using a snapshot of resource usage counters of $i^{th}$ worker node
using its model
Let $p \in [0, 1]$ be the minimum acceptable confidence of predictions.

 1: /* PREDICT runs every $\Delta$ interval in the background */
 2: procedure PREDICT
 3:     for all the workers in $N$
 4:         collect a snapshot of node's resource usage counters
 5:         $willStraggle_i$ = prediction if worker $n_i$ will create a straggler
 6:         $confidence_i$ = confidence in the above prediction

 7: procedure SCHEDULE
 8:     for a task chosen as per the preferred scheduling policy
 9:         when heartbeat is received from a worker indicating free slot(s)
10:             if $willStraggle_i$ == yes with $confidence_i > p$
11:                 reject the task from being assigned to $n_i$
12:             else
13:                 proceed as per the configured scheduling policy
```

3.5.1 *Wrangler's* Scheduling Algorithm

In Hadoop, the master manages job scheduling by assigning tasks to workers in response to *heartbeats* sent by them every few seconds. When the scheduler receives a heartbeat indicating that a map or reduce slot is free, the configured scheduling policy, such as Fair-scheduling, selects tasks to be assigned.

Wrangler's scheduling algorithm proposes to extend any of the existing schedulers. Before launching a task, our model-informed scheduler predicts if a worker will finish it in a timely manner. If the worker is predicted to create a straggler at that time, the task is not assigned to it. When we find a worker that is not predicted to create a straggler, the task is then assigned to it.

Algorithm 1 details this scheduling policy. The *predict* procedure (lines 2-6) is executed in background every Δ time interval to predict if the workers will create stragglers. All the predictions also have a confidence measure (line 6) attached to them. The *schedule* procedure (lines 7-13) is the hook we embed in the default scheduler code. We modified the Fair-scheduler code for our prototype (see Section 3.6). When a heartbeat is received, our scheduler delays a tasks's assignment to a worker if it is predicted to create a straggler with confidence higher than a configured threshold p in Algorithm 1. Otherwise, we let the default scheduling policy make the assignment decision (lines 12-13). Note that *Wrangler* processes the predictions in background and keeps them ready for the scheduler to use (see Section 3.6). This allows us to be off the critical path that makes scheduling decisions.

Note that *Wrangler* acts as a system that provides hints to the default/configured scheduler. This means that the decision of which task to launch next is left to the underlying scheduler. *Wrangler* only informs the scheduler whether or not a newly available node is likely to finish a task in timely manner. Also, this is an initial step; we left inclusion of task-level features for future work which could enable further improvements in overall job completion times.

3.5.2 Learning the Parameters: p and Δ

Algorithm 1 has two tunable parameters: p is the minimum acceptable confidence in predictions needed for them to influence the scheduling decisions and Δ, that decides how frequently a snapshot of node's resource usage counters is collected and predictions are computed using it. Next we explain how we tune these parameters.

3.5.2.1 Learning p: Threshold on Confidence Measure

Parameter p is the minimum acceptable confidence of predictions. p takes a value in range [0,1]. If p is too low, many tasks will get delayed, adversely affecting job completion and underutilizing resources. On the other hand, if p is too high, many long running tasks will get scheduled. We must set it to balance good resource utilization without increasing the chances of tasks straggling. We observed that the value of p needed for maximum improvement in overall job completion times varies from one workload to another.

Our approach is to learn p automatically during the model building phase so as to avoid the need for manual tuning. As we explained in Section 3.4.2.2, we use the data generated by tasks executed over initial 2 hours to build models and test their accuracy on the tasks submitted over the next hour. The set of tasks submitted in this one hour are unseen by the learning algorithm, referred to as a validation set. To decide the value of p, we use the confidence (see Section 3.4.1) on predictions on tasks in the validation set. For every node, we extracted a range of confidence values producing the most accurate predictions. We set p equal to the median of a range that is agreed upon by a majority of the nodes (forming a quorum). In Section 3.7.5, we show the sensitivity of *Wrangler* with respect to multiple values of p. In that section, we present the experimental validatation over the next 10 hours that the chosen values for p achieve maximum gain in job completion times for the workloads listed in Table 3.1. A different p value per node could also be used if desired.

3.5.2.2 Learning Δ: Interval between Predictions

Recall that to be off the critical path of making scheduling decisions, *Wrangler* keeps the predictions ready for the scheduler to use (Section 3.6 provides implementation details). To ensure that the predictions reflect current state of a node, *Wrangler* regularly collects nodes' resource usage counters and predicts using their respective models if a node can create a straggler at the time. Parameter Δ determines how frequently this background process should be invoked. If Δ is too high, the predictions may not be fresh enough to reflect dynamic changes in the node's status. Extracting predictions out of already built models is of the order of sub-milliseconds [31]; thus, it is not too expensive. Thus, small Δ is a safe choice. We suggest setting Δ to a smaller value than the minimum inter-task submission times.

The value we chose for our experimental setup was decided based on the time spent collecting the node resource usage counters and predicting based on them. With our current centralized implementation of *Wrangler's* prototype, where the resource usage counters from all the nodes are collected at the master using Ganglia, Δ is set to 15 seconds. On a distributed implementation of *Wrangler* (see Section 3.7.8), it is feasible to have every node collect its resource usage counters, predict using it and send this prediction to the master along with the heartbeat. This implementation makes Δ independent of the number of nodes in a cluster.

3.5.3 Bound on Delays Introduced by *Wrangler*

Since *Wrangler* delays tasks that may straggle, it is important to bound such a delay. Wrangler drives the cluster through the following three states (see Figure 3.7), with respect to the predictions on the constituent nodes:

- N: No node is predicted to create a straggler
- S: Some nodes are predicted to create stragglers and
- A: All the nodes are predicted to create stragglers.

For this analysis to be tractable, we make a simplifying assumption that the cluster behavior could be modeled as an irreducible ergodic Markov chain [87, 131] in which the cluster's future state depends on the currently observed state.

Let P be a 3×3 matrix describing the transition probabilities for the states in Figure 3.7. Let π be a 3×1 vector, $[\pi_N, \pi_S, \pi_A]^T$ comprising the steady state distributions of the 3 states. To ensure that *Wrangler* does not delay tasks indefinitely, we do not want the cluster to end up in state A at steady state. For this, we need to show that π_A is very close to 0. We do this analysis as follows. When the system attains steady state, the transition matrix P has no effect on the vector π. Using the elementary properties of stationary distributions, we have that $\pi P = \pi$ (for details, see [87]). This equation can be solved through eigen-analysis to find the eigenvector (π) of the transition matrix P corresponding to the eigenvalue of 1. Using the log of the state transitions of the cluster executing various workloads, we estimated π and found that π_A was indeed 0 as desired. In Section 3.7.7 we describe an empirical analysis of the delays induced by *Wrangler*. Here, we provided a theoretical guarantee that a task, as seen in real workloads, will never be delayed indefinitely. One way of bounding delays in case of under-provisioned systems or workloads with a high rate of job submission could be to provide a tunable parameter that limits the number of times a task's assignment gets rejected. We leave this for future work.

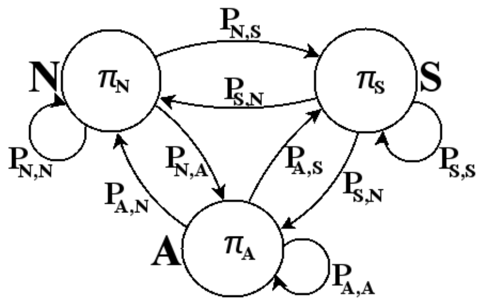


Figure 3.7: A simplified State-Machine that captures the behavior of a cluster executing parallel data intensive workloads.

3.6 Implementation of *Wrangler*

We implemented *Wrangler* (see Figure 3.1) by embedding it in the Fair scheduler's code. It consists of about 200 lines of code: about 10 lines embedded in Fair scheduler, and the rest for building models using SVM, capturing nodes' resource usage counters and extracting predictions from the models. We use Weka [83] for building SVM and logistic regression models. In our prototype, the Hadoop logs, node's resource usage counters are collected centrally at the master node for further processing and building models. However, *Wrangler* could be implemented in a distributed manner where all the worker nodes collect and process their statistics, build and use their models independently (see Section 3.7.8).

***Wrangler's* Training and Usage Workflow:** When a workload starts executing on the cluster, *Wrangler* collects the job logs and node-level statistics from all the nodes and processes them to generate the dataset to build models. It builds one straggler prediction model for each node in the cluster taking less then a second per node using Weka [83]. In our experiments, we captured training data for about 2 hours[2]. The jobs do not need to wait until the training period is over as the default scheduling policy will be in effect during this time. Once training data was collected, it took an order of a few seconds to build a model per node. Once the models are built, *Wrangler's* background process frequently captures the resource usage counters from all worker nodes. It predicts if a new task might run slower if assigned to a particular node using that node's model. *Wrangler* also reports the confidence it has in its prediction. Time taken by each prediction was the order of sub-milliseconds. This prediction for every worker node happens at a regular interval of Δ and is read by the scheduler for making assignment decisions. This way, the prediction process does not come in the critical path of making scheduling decisions.

Models could be updated or re-built regularly once enough jobs have finished execution and new data has been collected. This data collection and model building phase could overlap with job executions. We show the improvements achieved on the workloads from Facebook and Cloudera's customers, by building models only once and testing for the next 10 hours. We leave analyzing the

[2]We divided the data collected in these 2 hours in 3 parts; we used 2 of them for training and the remaining part for testing model's correctness.

usefulness of the training for continuously changing workloads as well as across different workloads for future work.

3.7 Evaluation of Wrangler

We demonstrate *Wrangler's* effectiveness by answering the following questions through experimental evaluation:

1. Does *Wrangler* improve job completion times?
2. Does *Wrangler* reduce resource consumption?
3. Is *Wrangler* reliable in the presence of modeling errors?
4. How sensitive is *Wrangler* with respect to parameters?
5. *How* does *Wrangler* improve job completion times?
6. What if *Wrangler* mis-predicts?
7. Does *Wrangler* scale well?

3.7.1 Setup

Real world production workloads: We evaluate using two workloads from Facebook (*FB2009* and *FB2010*) and two from Cloudera (*CC_b* and *CC_e*). We replay the production logs to synthesize representative workloads using SWIM [44] that faithfully reproduces the job submission patterns, data sizes, and data ratios on our 50-node cluster of m1.xlarge instances on Amazon EC2.

Baseline: Although *Wrangler* serves as a straggler avoidance approach on its own, it can also be used in conjunction with existing mitigation approaches to accrue further reduction in job completion times and resources consumed. To show its effectiveness, we compare *Wrangler* against speculative execution, a widely used straggler mitigation technique in Hadoop production clusters.

Metrics: We look at the % reduction in job completion times as our primary metric. Let T_w be the job execution time with *Wrangler* and T be without *Wrangler*, then

$$\%Reduction = \frac{T - T_w}{T} * 100 \tag{3.2}$$

A positive value indicates reduction in job completion times whereas negative values indicate increase.

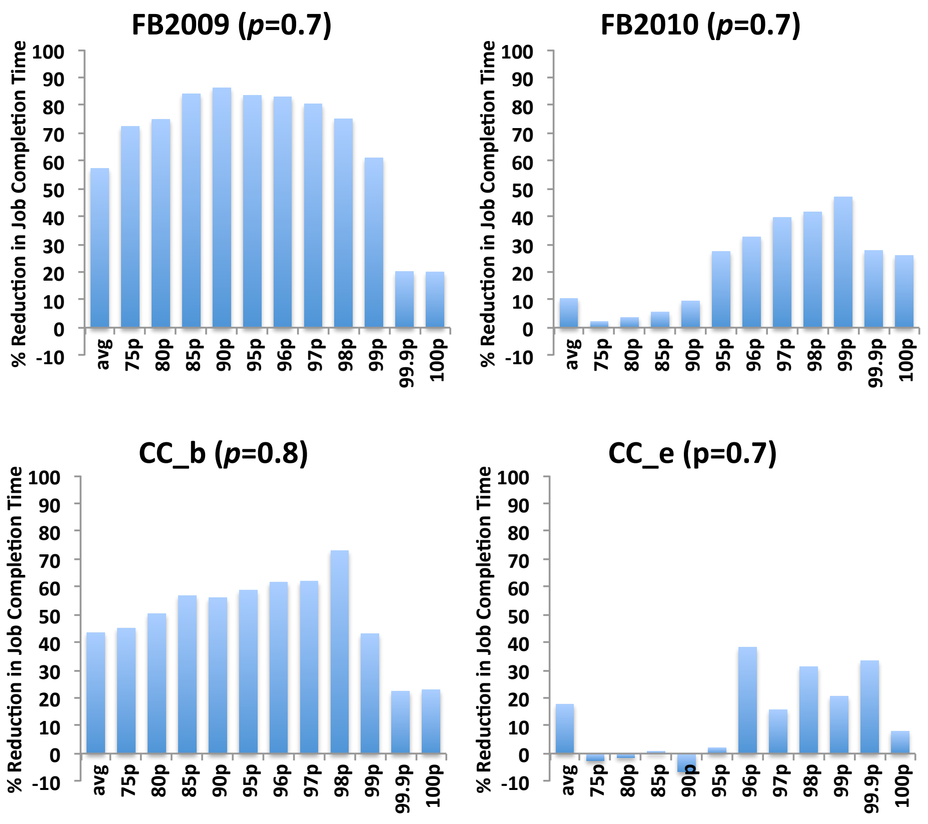


Figure 3.8: Summary of *Wrangler's* improvements in job completion times for all the workloads with the tuned vale of p (see Sections 3.5.2.1 and 3.7.5): This plot shows that *Wrangler* successfully reduces the completion time by carefully scheduling potential stragglers.

Highlights of our results are:

- For Facebook 2009 production Hadoop trace, *Wrangler* improves the overall job completion times by 61% at the 99^{th} percentile and by 20% at the 99.9^{th} percentile over Hadoop's speculative execution.
- For Cloudera customer's production Hadoop trace, *CC_b*, *Wrangler* improves the overall job completion times by 43% at the 99^{th} percentile and by 22% at the 99.9^{th} percentile over speculative execution.
- Being proactive, *Wrangler* consumed 55% and 40% lesser resources for Facebook 2009 and *CC_b* respectively compared to the reactive speculative execution mechanism.

Workload	Total Task-Seconds		% Reduction in
	w/o *Wrangler*	With *Wrangler*	total task-seconds
FB-2009	9,03,980	4,05,953	55.09
FB-2010	2,96,893	2,23,339	24.77
CC_b	2,01,444	1,20,559	40.15
CC_e	6,94,564	6,37,319	8.24

Table 3.2: Resource utilization with and without *Wrangler* in terms of total task execution times (in seconds) across all the jobs during our replay. Being proactive, *Wrangler* achieves the improved completion times while consuming lesser resources compared to the wait-and-speculate re-execution mechanisms.

3.7.2 Does *Wrangler* improve job completion times?

We evaluated the gains on the tasks of previously unseen jobs arriving after the models are built. Figure 3.8 shows improvement in average, 75^{th} and higher percentile job completion time statistics for the same set of jobs executed with *Wrangler* and without *Wrangler* (i.e., with speculative execution). For *FB2009*, *Wrangler* achieves an improvement of 61% at the 99^{th} percentile and 57% in the average job completion times. For CC_b, we see the improvements of 43% at the 99^{th} percentile and 44% in the average job completion times. Note that, for CC_e, *Wrangler* slowed down the jobs between 75^{th} and 90^{th} percentile. As per our analysis of this, we found that CC_e has a bursty job-submission pattern. This means that, in a time period shorter that Δ, many tasks were submitted. However, the resource usage counters from all the nodes were collected only once in this Δ time interval. The predictions were based on these resource usage counters and hence most likely were not timely representative of the load on the nodes. This could be avoided by setting Δ to a value smaller than the minimum inter-task submission times. Figure 3.8 summarizes the maximum gains achieved for each workload after tuning for the right value of p (Section 3.7.5).

For *FB2010* workload, we achieved lower gains compared to those achieved for *FB2009* workload. The number of stragglers found per hour in *FB2010* is comparable to those in *FB2009* workload. However, [43] notes that Facebook's workload has changed significantly from 2009 to 2010. *FB2010* has higher job submission rate, higher I/O rate and is more compute intensive. This does not affect the prediction accuracy, but the model-informed scheduling mechanism needs comparatively less-occupied nodes to achieve faster job completions. *Wrangler's* key idea is to avoid overloading a few nodes and instead distribute that load evenly. However, if load is consistently high on the cluster, *Wrangler's* gains are limited.

3.7.3 Does *Wrangler* reduce resource consumption?

We showed that *Wrangler* significantly improves the job completion time when used in conjunction with speculative execution. The reactive relaunch-based mechanism of speculative execution consumes extra resources for the redundantly launched copies of straggler tasks. On the contrary, being proactive *Wrangler* achieves overall faster job completions by smarter task to node assignments. Table 3.2 shows that *Wrangler* consumes lesser resources as compared to speculative execution resulting in reduced costs by freeing up the resources sooner. As we noted earlier, CC_e has a

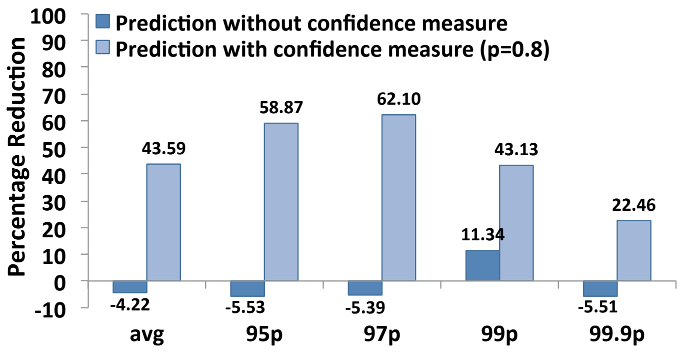


Figure 3.9: Confidence measures attached with the predictions are crucial. This plot compares the reduction in overall job completion time achieved by *Wrangler* with and without using the confidence measure attached with its predictions for CC_b workload.

bursty job submission pattern and even in this case, the delay based mechanism of *Wrangler* speeds up the jobs while using lesser total resources.

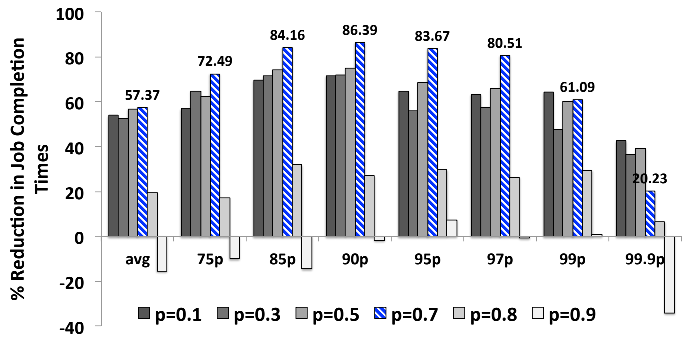
3.7.4 Is *Wrangler* reliable in presence of modeling errors?

Wrangler's achieves reliability in presence of prediction errors by attaching confidence with the straggler predictions. We have observed that this confidence measure plays a crucial role in improving the completion times for all the workloads by allowing only confident predictions to influence scheduling decisions. Figure 3.9 shows the percentage reduction in job completion times achieved by *Wrangler* with and without confidence measures for the *CC_b* workload as an example. In the absence of confidence measures, the modeling errors drive the scheduling decisions to change too frequently. This makes the costs incurred by delaying the tasks to weigh more than the reduction achieved in job completion times. Thus, *Wrangler* achieves its goal of being robust to modeling errors by the novel use of confidence measures.

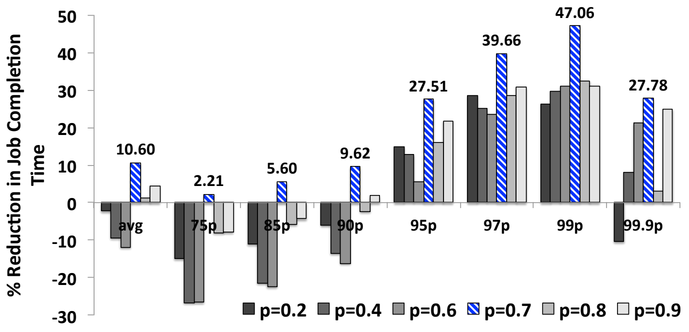
3.7.5 How sensitive is *Wrangler* with respect to p?

We described how to learn a value of p during training in Section 3.5.2.1. In this section, we evaluate the sensitivity of *Wrangler* with respect to p. We calculated the percentage reduction in job completion times *Wrangler* achieves with different values of p in a range of $[0, 1]$. In Figure 3.10, we present a subset of them. Our experimental validation shows that for both the Facebook workloads (*FB2009* and *FB2010* in Figure 3.10 (a) and (b)) a value of 0.7 attains the maximum gains in terms of improved job completion times and resource utilization. The right value of p turned out to be 0.8 for *CC_b* and 0.7 for *CC_e* (see Figure 3.10 (c) and (d) respectively.

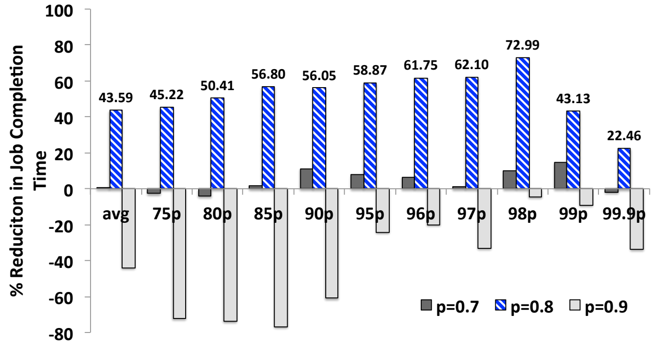
From this analysis, we note that choosing the right value for p is important for improving job completion times. We also note that this value is workload dependent. Small changes in its value could change the improvements drastically for some workloads (for example, see CC_b in Figure 3.10 (c)). However, in Section 3.5.2.1 we described an automatic way to choose a starting



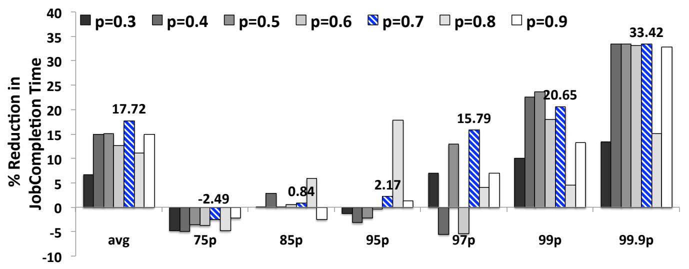
(a) FB2009



(b) FB2010



(c) CC_b



(d) CC_e

Figure 3.10: Reduction in job completion times achieved by *Wrangler* with various values of p for all the workloads. Data labels are shown for the p value that achieves the highest gain in completion times for each of the four workloads.

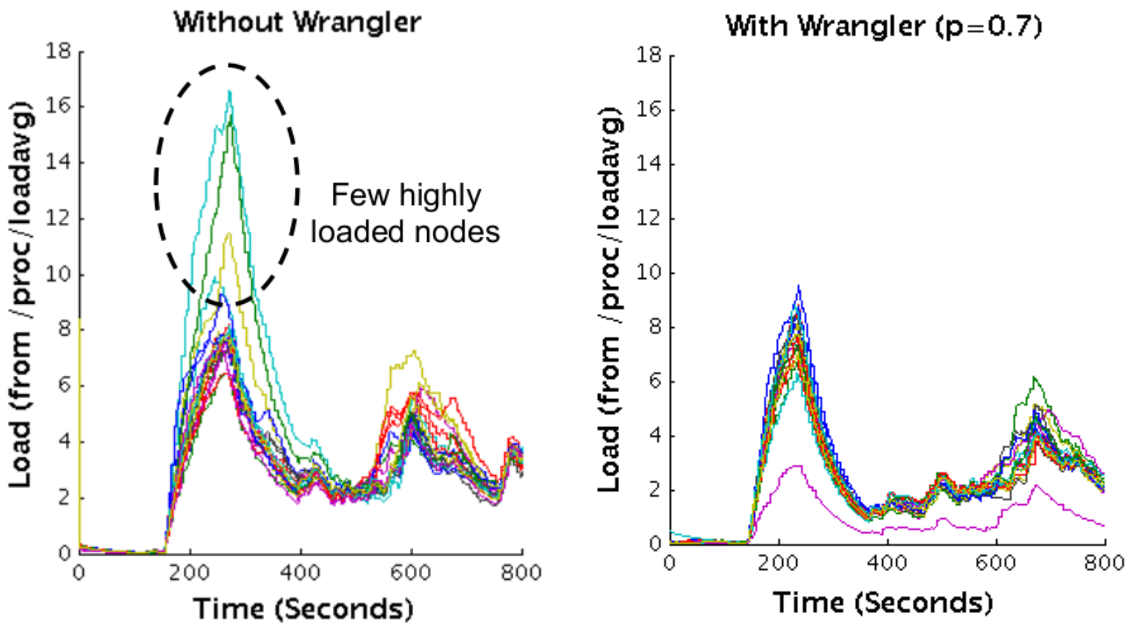


Figure 3.11: Load/resource utilization without and with *Wrangler* (p=0.7) for *FB2010* workload: Even with highly intensive *FB2010* workload, *Wrangler* speeds up the 99^{th} percentile of job completions by 47% by avoiding overloading of a few nodes and distributing the load more evenly (see Section 3.7.6).

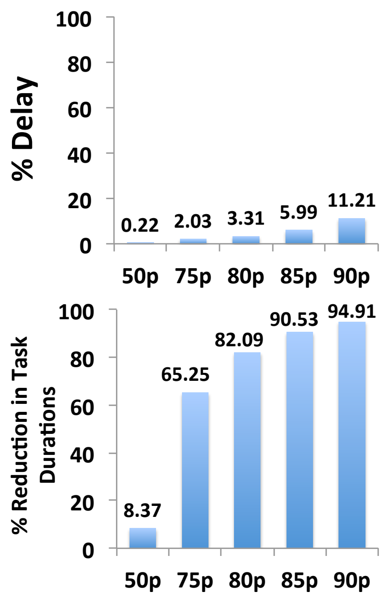
value for p during the model building phase itself without waiting for jobs to execute. Additionally, to tune to its right value, we need to observe the performance of the jobs for relatively small amount of time (we verified on a window of 30 minutes). *Wrangler's* architecture is flexible and allows changes in value of p on the fly until it converges to a right value.

3.7.6 *How* does *Wrangler* improve job completion times?

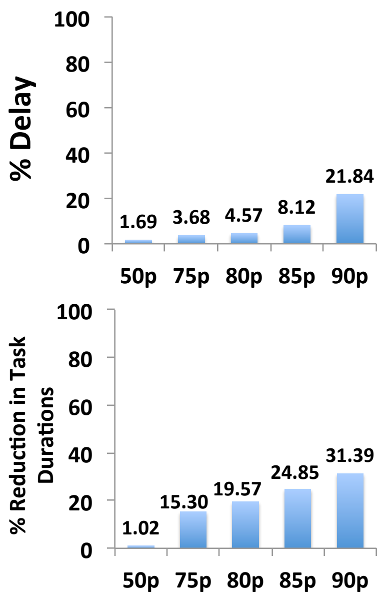
Wrangler avoids stragglers by implicitly doing a better job of load balancing than the scheduling approaches that use only statically available information. Figure 3.11 shows the load on the cluster executing the *FB2010* workload without *Wrangler* and with *Wrangler* (p=0.7). Without *Wrangler*, a few nodes enter a heavily loaded state and tend to remain loaded as new tasks are then assigned to them in addition. For a right value of p, with *Wrangler*, we observed that the load is now distributed evenly and most of the nodes are almost equally utilized. So even in this limiting case of *FB2010*, where the cluster is heavily loaded, *Wrangler* improves job completions significantly (47.06% at the 99^{th} percentile, see Figure 3.10 (b)).

3.7.7 What if *Wrangler* mis-predicts?

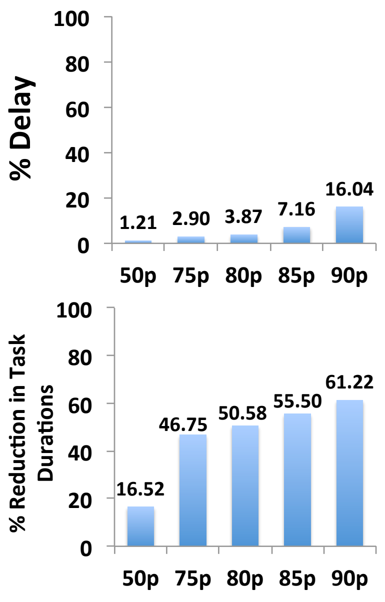
The main impact of wrong predictions is on the delay experienced by tasks. In Section 3.5.3, we statistically proved that *Wrangler* will not delay tasks indefinitely. Also, since our method only



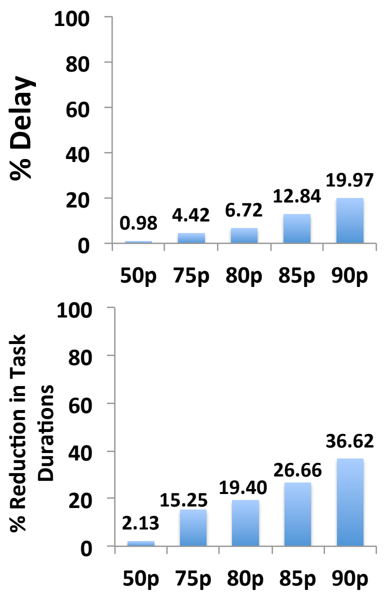
(a) FB2009



(b) FB2010

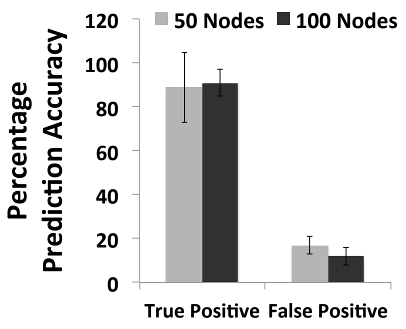


(c) CC_b

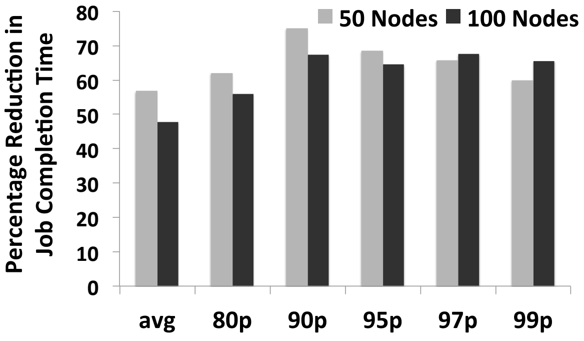


(d) CC_e

Figure 3.12: Empirical analysis of the amount of delay introduced by *Wrangler* and the speed up achieved in task completions: With marginal delays, *Wrangler* achieves significant reduction in task durations. *FB2010* being a resource intensive workload [43], *Wrangler* induces slightly higher delays seeking for nodes that are not over-committed. This results in reduced improvements since the cluster is mostly loaded. Since *CC_e* contains lesser number of stragglers, *Wrangler* has limited opportunity to improve task completions (see Section 3.7.7).



(a) Prediction Accuracy



(b) Improved Job Completion

Figure 3.13: Scalability of Wrangler's centralized prototype: The prediction accuracies shown in (a) and percentage reduction in job completion times shown in (b) for a larger (100 nodes) cluster are significant and comparable with those for a 50 nodes cluster.

delays potential stragglers, there is no risk of terminating or replacing them. Thus, *Wrangler* does not impact the correctness or liveness properties of the system.

In this section, we analyze empirically observed delays for the Facebook and Cloudera workloads in our dataset. Figure 3.12 shows the percentage delay with respect to the task's original execution time (i.e., without *Wrangler*) and the percentage reduction in *task* completion times achieved by *Wrangler* for these workloads. Note that Figure 3.12 presents speed-up at *task*-level whereas all the other results presented earlier are the *job*-level speed-ups. We see that for *FB2009*, the delay introduced by *Wrangler* is less than 2% for 75% of the tasks and less than 11% for 90% of the tasks. Notice that the delays introduced for *FB2010* workload are higher than *FB2009* and the corresponding improvements in task completion times are also lower. This is due to the high load on all the nodes of the cluster executing *FB2010* as we described in Section 3.7.2. This explains *Wrangler's* decision to delay the tasks more while seeking for situations where the nodes are not over committed.

3.7.8 Does *Wrangler* scale well?

Our current prototype of *Wrangler* is centralized: the resource usage counters from all the nodes are collected, processed, node-wise models are built and used centrally from a location accessible to the scheduler. Most actual users use a cluster size of about 100 nodes [101, 111, 7]. We executed the $FB2009$ workload on a larger cluster with 100 nodes. Note that in our experiments, we maintain fixed amount of work per node by re-scaling the real-life workload appropriately.

Figure 3.13 (a) shows that we get comparable prediction accuracies of the models for the small and larger clusters. Moreover, Figure 3.13 (b) shows that the percentage reduction in job completion time achieved by *Wrangler's* centrally implemented prototype is comparable across cluster sizes.

Results from Figures 3.13 (a) and (b) show that *Wrangler* scales well to larger cluster sizes. The central implementation might stress the master with increase in the size of a cluster further. We believe that *Wrangler* could be extended for even larger clusters by distributing these responsibilities to individual nodes. This allows for most of the computations to take place locally, thereby reducing the network traffic. In this scenario, each node would build its own model using locally collected resource usage counters. At a regular time interval Δ (see Section 3.5.2.2), the straggler predictions would be computed using locally available recent resource usage counters and model at each node. These model predictions would then be piggybacked with the heartbeat messages sent to the master by each node. Finally, the master would use these predictions to influence scheduling decisions as described in our current workflow. We leave this as an avenue for future research.

3.8 Conclusion

Wrangler proactively avoids stragglers to achieve faster job completions while using fewer resources. Rather than allowing tasks to execute and detecting them as stragglers when they run slow, *Wrangler* predicts stragglers before they are launched. *Wrangler's* notion of *confidence measures* allows it to overcome modeling errors. Further, *Wrangler* leverages these confidence measures to achieve a reliable task scheduling; thus eliminating the need for replicating them. Prototype on Hadoop using an EC2 cluster of 50 nodes showed that *Wrangler* speeds up the 99^{th} percentile job execution times by up to 61% and consumes up to 55% fewer resources as compared to the speculative execution for production workloads at Facebook and Cloudera's customers. Although it serves as a straggler avoidance approach on its own, *Wrangler* can also be used in conjunction with existing mitigation approaches. As a next step, we aim at speeding up the training process by (i) reducing the time spent for capturing training data per node in a cluster and (ii) by training straggler prediction models across workloads.

Chapter 4

Fast Training: Building Data-Driven Models using Multi-Task Learning

In the previous chapter, we demonstrated how we built data-driven models leveraging performance and utilization data that enable significant improvement in job tail latencies. Several other relevant works have applied machine learning techniques to different systems problems [79, 57, 58, 32, 25]. Availability of large and representative training data is the fundamental requirement for building accurate models. Collecting such a training dataset can be time-consuming. In some cases to ensure representative training data is collected, we may have to let systems run into undesirable states, such as degraded performance or poor utilization. Thus, adoption of data-driven models in real-world systems faces the challenge of cost for training data collection.

To address this challenge, in this chapter, we propose multi-task learning formulations that share information between the various models, allowing us to use less training data and bring training time significantly down. We use our predictive scheduler, Wrangler, as an example and build on it to formulate, implement, and evaluate our multi-task learning formulations that capture the shared structure in our data, improving generalization performance on limited data.

4.1 Shortcomings of prior work and avenues for improvements

As we described in the previous chapter, due to the heterogeneity of nodes in a cluster, Wrangler's model builder trains a separate classifier for each node. Note that to build a training set per node, every node should have executed a sufficient number of tasks. Wrangler takes a few hours (approximately 2-4 hours, depending on the workload) for this process. Additionally, because each workload might be different, these models are retrained for every new workload. Thus, for every new workload that is executed on the cluster, there is a 2-4 hour model building period. In typical large production clusters with tens of thousands of nodes, it might be a long time before a node collects enough data to train a classifier. Later in Table 4.4 we show that the prediction accuracy of

Wrangler rapidly degrades as the amount of training data is reduced. In some cases, workloads may only be run a few times in total, limiting the ability of systems like Wrangler to make meaningful predictions.

Alternatively, large clusters with locality aware scheduling can lead to poor sample coverage. Recall that, in our case, each task of a workload executed on a node amounts to a training data point. The placement of input data on nodes in a cluster is managed by the underlying distributed file system [70]. To achieve locality for faster reading of input data, sophisticated locality-aware schedulers [170, 168] try to assign tasks to nodes already having the appropriate data. Based on the popularity of the data, number of tasks assigned to a node could vary. Hence, we may not get uniform number of training data points, i.e., tasks executed, across all the nodes in a cluster. There could be other reasons behind skewed assignment of tasks to nodes [99]: even when every map task has the same amount of data, a task may take longer depending on the code path it takes based on the data it processes. Hence, the node slots will be busy due to such long running tasks. This could lead to some nodes executing fewer tasks than others.

These shortcomings can be addressed if each classifier is able to leverage information gleaned at other nodes and from other workloads. For instance, when there is not enough data at a node for a workload, we can gain from the data collected at that node while it was executing other workloads, or from other nodes running the same workload. Such information sharing falls in the ambit of multi-task learning (MTL), where the learner is embedded in an environment of related tasks, and the learner's aim is to leverage similarities between the tasks to improve performance of all tasks.

In applications such as ours, high accuracy is not the only objective: we also want to be able to gain some insight into what is causing these stragglers. Such insight can aid in debugging root causes of stragglers and system performance degradation. Automatically learned classifiers can be opaque and hard to interpret.

These observations suggest that our modeling framework needs to be robust to limited and potentially skewed data. Thus, there is a need for straggler prediction models (1) that can be trained using minimum available data, (2) that generalize to unseen nodes or workloads, and (3) that allow us to gain insights into what is causing these stragglers.

In the following section, we describe our multi-task learning based approach with these goals for avoiding stragglers. To enable us to gain insights into models' learning, we propose group sparsity inducing mixed-norm regularization on top of our basic MTL formulation to automatically reveal the group structure that exists in the data, and hopefully provide insights into how straggler behavior is related across nodes and workloads. We also explore sparsity-inducing formulations based on feature selection that attempt to reveal the characteristics of straggler behavior. We evaluate it using real world production level traces from Facebook and Cloudera's customers, and compare the gains with Wrangler.

4.2 Multi-task learning for straggler avoidance

As described in the previous section, Wrangler builds separate models for each workload and for every node. Thus, every {node, workload} tuple is a separate learning problem. However, learning problems corresponding to different workloads executed on the same node clearly have something in common, as do learning tasks corresponding to different nodes executing the same workload. We want to use this shared structure between the learning problems to reduce data collection time.

Concretely, a task executing on a node will be a straggler because of a combination of factors. Some of these factors involve the properties of the node where the task is executing (for instance, the node may be memory-constrained) and some others involve particular requirements that the tasks might have in terms of resources (for instance, the task may require a lot of memory). These are workload-related factors. When collecting data for a new workload executing on a given node, one must be able to use information about the workload collected while it executed on other nodes, and information collected about the node collected while it executed other workloads.

We turn to multi-task learning to leverage this shared structure. In the terminology of multi-task learning, each $\{$node, workload$\}$ pair forms a separate learning-task[1] and these learning problems have a shared structure between them. However, unlike typical MTL formulations, our learning-tasks are not simply correlated with each other; they share a specific structure, clustering along node- or workload-dependent axes. In what follows, we first describe a general MTL formulation that can capture such learning-task grouping. We then detail how we apply this formulation to our application of straggler avoidance.

4.2.1 Partitioning tasks into groups

Suppose there are T learning tasks, with the training set for the t-th learning-task denoted by $D_t = \{(\mathbf{x}_{it}, y_{it}) : i = 1, \ldots, k_t\}$, with $\mathbf{x}_{it} \in \mathbb{R}^d$. We begin with the formulation proposed by [61]. Evgeniou, et al. proposed a basic hierarchical regression formulation with the linear model, $\mathbf{w}_t$, for learning-task t as:

$$\mathbf{w}_t = \mathbf{w}_0 + \mathbf{v}_t \tag{4.1}$$

The intuition here is that $\mathbf{w}_0$ captures properties common to all learning-tasks, while $\mathbf{v}_t$ captures how the individual learning-tasks deviate from $\mathbf{w}_0$.

We then frame learning in the context of empirical loss minimization. Given the variability in node behavior and the need for robust predictions we adopt the hinge-loss and apply l_2 regularization to both the base w_0 and learning-task specific weights v_t:

$$\min_{\mathbf{w}_0, \mathbf{v}_t, b} \lambda_0 \|\mathbf{w}_0\|^2 + \frac{\lambda_1}{T} \sum_{t=1}^{T} \|\mathbf{v}_t\|^2 + \sum_{t=1}^{T} \sum_{i=1}^{k_t} L_{Hinge}\left(y_{it}, (\mathbf{w}_0 + \mathbf{v}_t)^T \mathbf{x}_{it} + b\right) \tag{4.2}$$

where $L_{Hinge}(y_{it}, s_{it}) = \max(0, 1 - y_{it} s_{it})$ is the hinge loss.

In the above formulation, all learning-tasks are treated equivalently. However, as discussed above, in our application some learning-tasks naturally cluster together. Suppose that the learning-tasks cluster into G non-overlapping *groups*, with the t-th learning-task belonging to the $g(t)$-th group. Note that while we derive our formulations for non-overlapping groups, which is true in

[1]The machine learning notion of a "task" as a learning problem differs from the cloud computing notion of a "task" as a part of a job that is run in parallel. The intended meaning should be clear from the context.

our application, the modification for overlapping groups is trivial. Using the same intuition as Equation 4.1, we can write the classifier $\mathbf{w}_t$ as:

$$\mathbf{w}_t = \mathbf{w}_0 + \mathbf{v}_t + \mathbf{w}_{g(t)} \tag{4.3}$$

In general, there may be more than one way of dividing our learning-tasks into groups. In our application, one may split learning-tasks into groups based on workload or based on nodes. We call one particular way of dividing learning-tasks into groups a *partition*. The p-th partition has G_p groups, and the learning-task t belongs to the $g_p(t)$ group under this partition. Now, we also have a separate set of weight vectors for each partition p, and the weight vector of the g-th group of the p-th partition is denoted by $\mathbf{w}_{p,g}$. Then, we can write the classifier $\mathbf{w}_t$ as:

$$\mathbf{w}_t = \mathbf{w}_0 + \mathbf{v}_t + \sum_{p=1}^{P} \mathbf{w}_{p,g_p(t)} \tag{4.4}$$

Finally, note that $\mathbf{w}_0$ and $\mathbf{v}_t$ can also be seen as weight vectors corresponding to trivial partitions: $\mathbf{w}_0$ corresponds to the partition where all learning-tasks belong to a single group, and $\mathbf{v}_t$ corresponds to the partition where each learning-task is its own group. Thus, we can include $\mathbf{w}_0$ and $\mathbf{v}_t$ in our partitions and write Equation 4.4 as:

$$\mathbf{w}_t = \sum_{p=1}^{P} \mathbf{w}_{p,g_p(t)} \tag{4.5}$$

Intuitively, at test time, we get the classifier for the t-th learning-task by summing weight vectors corresponding to each group to which t belongs.
As in Equation 4.2, the learning problem involves minimizing the sum of l_2 regularizers on each of the weight vectors and the hinge loss:

$$\boxed{\min_{\mathbf{w}_{p,g},b} \sum_{p=1}^{P} \sum_{g=1}^{G_p} \lambda_{p,g} \|\mathbf{w}_{p,g}\|^2 + \sum_{t=1}^{T} \sum_{i=1}^{k_t} L_{Hinge}\left(y_{it}, \left(\sum_{p=1}^{P} \mathbf{w}_{p,g_p(t)} \right)^T \mathbf{x}_{it} + b \right) \quad (4.6)}$$

Here, the regularizer coefficient $\lambda_{p,g} = \frac{\lambda_p \#(p,g)}{T}$, where $\#(p,g)$ denotes the number of learning-tasks assigned to the g-th group of the p-th partitioning. The scaling factor $\frac{\lambda_p \#(p,g)}{T}$ interpolates smoothly between λ_0, when all learning-tasks belong to a single group, and $\frac{\lambda_1}{T}$, when each learning-task is its own group. Lowering λ_p for a particular partition will reduce the penalty on the weights of the p-th partition and thus cause the model to rely more on the p-th partitioning. For the base partition $p = 0$, setting λ_p = 0 would thus favor as much parameter sharing as feasible.

4.2.2 Reduction to a standard SVM

One advantage of the formulation we use is that it can be reduced to a standard SVM [51], allowing the usage of off-the-shelf SVM solvers. Below, we show how this reduction can be achieved. Given

λ, for every group g of every partition p, define:

$$\tilde{\mathbf{w}}_{p,g} = \sqrt{\frac{\lambda_{p,g}}{\lambda}}\mathbf{w}_{p,g} \tag{4.7}$$

Now concatenate these vectors into one large weight vector $\tilde{\mathbf{w}}$:

$$\tilde{\mathbf{w}} = [\tilde{\mathbf{w}}_{1,1}^T, \ldots, \tilde{\mathbf{w}}_{p,g}^T, \ldots, \tilde{\mathbf{w}}_{P,G_P}^T]^T \tag{4.8}$$

Then, it can be seen that $\lambda\|\tilde{\mathbf{w}}\|^2 = \sum_{p=1}^{P}\sum_{g=1}^{G_p}\lambda_{p,g}\|\mathbf{w}_{p,g}\|^2$. Thus, with this change of variables, the regularizer in our optimization problem resembles a standard SVM. Next, we transform the data points $\mathbf{x}_{it}$ into $\phi(\mathbf{x}_{it})$ such that we can replace the scoring function with $\tilde{\mathbf{w}}^T\phi(\mathbf{x}_{it})$. This transformation is as follows. Again, define:

$$\phi_{p,g}(\mathbf{x}_{it}) = \delta_{g_p(t),g}\sqrt{\frac{\lambda}{\lambda_{p,g}}}\mathbf{x}_{it} \tag{4.9}$$

Here, $\delta_{g_p(t),g}$ is a Kronecker delta, which is 1 if $g_p(t) = g$ (i.e. , if the learning-task t belongs to group g in the p-th partitioning) and 0 otherwise. Our feature transformation is then the concatenation of all these vectors:

$$\phi(\mathbf{x}) = [\phi_{1,1}(\mathbf{x})^T, \ldots, \phi_{p,g}(\mathbf{x})^T, \ldots, \phi_{P,G_P}(\mathbf{x})^T]^T \tag{4.10}$$

It is easy to see that:

$$\tilde{\mathbf{w}}^T\phi(\mathbf{x}_{it}) = \left(\sum_{p=1}^{P}\mathbf{w}_{p,g_p(t)}\right)^T\mathbf{x}_{it} \tag{4.11}$$

Intuitively, $\tilde{\mathbf{w}}$ concatenates all our parameters with their appropriate scalings into one long weight vector, with one block for every group of every partitioning. $\phi(\mathbf{x}_{it})$ transforms a data point into an equally long feature vector, by placing scaled copies of $\mathbf{x}_{it}$ in the appropriate blocks and zeros everywhere else.

With these transformations, we can now write our learning problem as :

$$\min_{\tilde{\mathbf{w}},b}\lambda\|\tilde{\mathbf{w}}\|^2 + \sum_{t=1}^{T}\sum_{i=1}^{k_t}L_{Hinge}(y_{it}, \tilde{\mathbf{w}}^T\phi(\mathbf{x}_{it}) + b) \tag{4.12}$$

which corresponds to a standard SVM. In practice, we use this transformation and change of variables, both at train time and at test time.

4.2.3 Automatically selecting groups or partitions

In many real world applications including our own, good classification accuracy is not an end in itself. Model interpretability is of critical importance. A large powerful classifier working on tons of features may do a very good job of classification, but will not provide any insight to an engineer trying to identify the root causes of a problem. It will also be difficult to debug such a classifier if and when it fails. In the absence of sufficient training data, large models may also overfit. Thus, interpretability and simplicity of the model are important goals. In the context of the formulation above, this means that we want to keep only a minimal set of groups/partitions while maintaining high classification accuracy.

We can use mixed l_1 and l_2 norms to induce a sparse selection of groups or partitions [21]. Briefly, suppose we are given a long weight vector $\mathbf{w}$ divided into M blocks, with the m-th block denoted by $\mathbf{w}_m$. Then the squared mixed l_1 and l_2 norm is:

$$\Omega(\mathbf{w}) = \left(\sum_{m=1}^{M} \|\mathbf{w}_m\|_2 \right)^2 \tag{4.13}$$

This can be considered as an l_1 norm on a vector with elements $\|\mathbf{w}_m\|_2$. When $\Omega(\cdot)$ is used as a regularizer, the l_1 norm will force some elements of this vector to be set to 0, which in turn would force the corresponding *block* of weights $\mathbf{w}_m$ to be set to 0. It thus encourages entire blocks of the weight vector to be set to 0, a sparsity pattern that is often called "group sparsity".

In our context, our weight vector $\tilde{\mathbf{w}}$ is made up of $\tilde{\mathbf{w}}_{p,g}$. We consider two alternatives:

1. We can put all the weight vectors corresponding to a single partition in the same block. Then, the regularizer becomes:

$$\boxed{\Omega_{ps}(\tilde{\mathbf{w}}) = \left(\sum_{p=1}^{P} \|\tilde{\mathbf{w}}_p\|_2 \right)^2 \tag{4.14}}$$

   where $\tilde{\mathbf{w}}_p = [\tilde{\mathbf{w}}_{p,1}^T, \ldots, \tilde{\mathbf{w}}_{p,G_p}^T]^T$ concatenates all weight vectors corresponding to the p-th partition. This regularizer will thus tend to kill entire partitions. In other words, it will uncover notions of grouping, or learning-task similarity, that are the most important.

2. We can put the weight vector of each group of each partition in separate blocks. Then the regularizer becomes:

$$\boxed{\Omega_{gs}(\tilde{\mathbf{w}}) = \left(\sum_{p=1}^{P} \sum_{g=1}^{G} \|\tilde{\mathbf{w}}_{p,g}\|_2 \right)^2 \tag{4.15}}$$

   This regularizer will tend to select a small set of groups which are needed to get a good performance.

4.2.4 Automatically selecting features

Mixed norms can also be used to select a sparse set of feature blocks that are most useful for classification. This is again useful for interpretability. In our application, features are resource counters at a node. Some of these features are related to memory, others to cpu usage, etc. If our model is able to predict straggler behavior solely on the basis of memory usage counters, for instance, then it might indicate that the cluster or the workload is memory constrained.

In our multi-task setting, such feature selection can also interact with the different groups and partitions of learning-tasks. Suppose each feature vector $\mathbf{x}$ is made up of blocks corresponding to different kinds of features, denoted by $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots \mathbf{x}^{(B)}$. We can similarly divide each weight vector $\tilde{\mathbf{w}}_{p,g}$ into blocks denoted by $\tilde{\mathbf{w}}_{p,g}^{(1)}$, $\tilde{\mathbf{w}}_{p,g}^{(2)}$ and so on. Then we have two alternatives.

1. One can concatenate corresponding feature blocks from all the weight vectors to get $\tilde{\mathbf{w}}^{(1)}, \ldots, \tilde{\mathbf{w}}^{(B)}$. Then a mixed l_1 and l_2 regularizer using these weight blocks can be written as:

$$\boxed{\Omega_{fs1}(\tilde{\mathbf{w}}) = \left(\sum_{b=1}^{B} \|\tilde{\mathbf{w}}^{(b)}\|_2\right)^2} \tag{4.16}$$

   Such a regularizer will encourage the model to select a sparse set of feature blocks on which to base its decision, setting *all* weights corresponding to all the other feature blocks to 0.

2. An alternative would be to let each group vector choose its own sparse set of feature blocks. This can be achieved with the following regularizer:

$$\boxed{\Omega_{fs2}(\tilde{\mathbf{w}}) = \left(\sum_{p,g}\sum_{b=1}^{B} \|\tilde{\mathbf{w}}_{p,g}^{(b)}\|_2\right)^2} \tag{4.17}$$

4.2.5 Kernelizing the formulation

We have till now described our formulation in primal space. However, kernelizing this formulation is simple. First, note that if we use a simple squared l_2 regularizer, then our formulation is equivalent to a standard SVM in a transformed feature space. This transformed feature space corresponds to a new kernel:

$$\begin{aligned} K_{new}(\mathbf{x}_{it}, \mathbf{x}_{ju}) &= \langle \phi(\mathbf{x}_{it}), \phi(\mathbf{x}_{ju}) \rangle \\ &= \sum_{p,g} \langle \phi_{p,g}(\mathbf{x}_{it}), \phi_{p,g}(\mathbf{x}_{ju}) \rangle \\ &= \sum_{p,g} \delta_{g_p(t),g}\delta_{g_p(u),g} \frac{\lambda}{\lambda_{p,g}} K(\mathbf{x}_{it}, \mathbf{x}_{jt}) \end{aligned} \tag{4.18}$$

Thus, the transformed kernel corresponds to the linear combination of a set of kernels, one for each group. Each group kernel is zero unless both data points belong to the group, in which case it equals a scaled version of the kernel in the original feature space.

When using the mixed-norm regularizers, the derivation is a bit more involved. We first note that [2]:

$$\left(\sum_{m=1}^{M} \|\mathbf{w}_m\|_2\right)^2 = \min_{0\leq\eta\leq 1, \|\eta\|_1\leq 1} \sum_{m=1}^{M} \frac{\|\mathbf{w}_m\|_2^2}{\eta_m} \tag{4.19}$$

Using this transformation leads to the following primal objective:

$$\min_{\mathbf{w}, b, 0\leq\eta\leq 1, \|\eta\|_1\leq 1} \sum_{m=1}^{M} \frac{\|\mathbf{w}_m\|_2^2}{\eta_m} + \sum_{i,t} L(x_{it}, y_{it}, \mathbf{w}) \tag{4.20}$$

where $L(x_{it}, y_{it}, \mathbf{w})$ is the hinge loss. This corresponds to a 1-norm multiple kernel learning formulation [95] where each block of feature weights corresponds to a separate kernel. We refer the reader to [95] for a description of the dual of this formulation.

We note that these kernelized versions are very similar to the ones derived in [156, 27]. However, the group and partition structure and the application domain are unique to ours.

4.2.6 Application to straggler avoidance

We apply this formulation to straggler avoidance as follows. Suppose there are N nodes and L workloads. Then there are $N \times L$ learning-tasks, and Wrangler trains as many models, one for each $\{$node, workload$\}$ tuple. For our proposal, we consider four different partitions:

1. A single group consisting of all nodes and workloads. This gives us the single weight vector $\mathbf{w}_0$.

2. One group for each node, consisting of all L learning-tasks belonging to that node. This gives us one weight vector for each node $\mathbf{w}_n, n = 1, \dots, N$, that captures the heterogeneity of nodes.

3. One group for each workload, consisting of all N learning-tasks belonging to that workload. This gives us one weight vector for each workload $\mathbf{w}_l, l = 1, \dots, L$, that captures peculiarities of a particular workload.

4. Each $\{$node, workload$\}$ tuple as its own group. Since there are $N \times L$ such pairs, we get $N \times L$ weight vectors, which we denote as $\mathbf{v}_t$, following the notation considered in [61].

Thus, if we use all four partitions, the weight vector $\mathbf{w}_t$ for a given workload l_t and a given node n_t is:

$$\mathbf{w}_t = \mathbf{w}_0 + \mathbf{w}_{n_t} + \mathbf{w}_{l_t} + \mathbf{v}_t \tag{4.21}$$

Figure 4.1 shows an example. The learning problem for the FB2009[2] workload running on node 1 belongs to one group from each of the four partitions mentioned above: (1) the global partition,

[2]See Section 4.3.1 for details about the workloads we use.

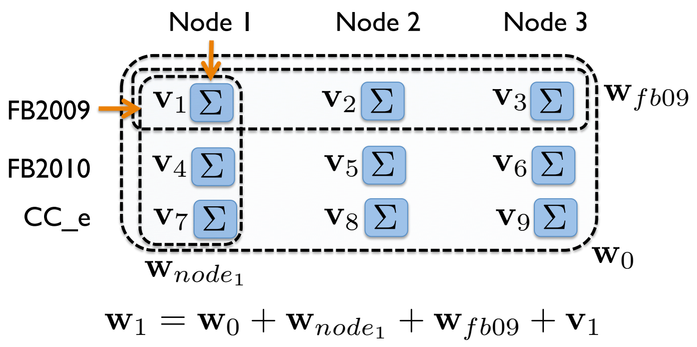


Figure 4.1: In our context of straggler avoidance, the learning-tasks naturally cluster into various groups in multiple partitions. When a particular learning- task, for example, node 1 and workload FB2009 ($\mathbf{v}_1$), has limited training data available, we learn its weight vector, $\mathbf{w}_1$, by adding the weight vectors of groups it belongs to from different partitions.

denoted by the weight vector, $\mathbf{w}_0$, (2) the group corresponding to node 1 from the node-wise partition, denoted by the weight vector $\mathbf{w}_{node_1}$, (3) the group corresponding to the FB2009 workload from the workload-wise partition, denoted by the weight vector $\mathbf{w}_{fb09}$, and (4) the group containing just this individual learning problem, denoted by the weight vector $\mathbf{v}_1$. Thus, we can learn the weight vector $\mathbf{w}_1$ as:

$$\mathbf{w}_1 = \mathbf{w}_0 + \mathbf{w}_{node_1} + \mathbf{w}_{fb09} + \mathbf{v}_1 \tag{4.22}$$

The corresponding training problem is then:

$$\boxed{\begin{aligned} \min_{\mathbf{w},b} \lambda_0 \|\mathbf{w}_0\|^2 + \frac{\nu}{N}\sum_{n=1}^{N}\|\mathbf{w}_n\|^2 + \frac{\omega}{L}\sum_{l=1}^{L}\|\mathbf{w}_l\|^2 + \frac{\tau}{T}\sum_{t=1}^{T}\|\mathbf{v}_t\|^2 \\ + \sum_{t=1}^{T}\sum_{i=1}^{k_t} L_{Hinge}\left(y_{it}, (\mathbf{w}_0 + \mathbf{w}_{n_t} + \mathbf{w}_{l_t} + \mathbf{v}_t)^T \mathbf{x}_{it} + b\right) \qquad (4.23)\end{aligned}}$$

where λ_0, ν, ω, τ are hyperparameters. We ran an initial grid search on a validation set to fix these hyperparameters and found that prediction accuracy was not very sensitive to these settings: several settings gave very close to optimal results. We used $\lambda_0 = \nu = \omega = \tau = 1$, which was one of the highest performing settings, for all our experiments. Appendix 4.6 provides the details of this grid search experiment along with the sensitivity of these hyperparameters to a set of values. As described in Section 4.2.2, we work in a transformed space in which the above problem reduces to a standard SVM. In this space the weight vector is

$$\tilde{\mathbf{w}} = [\tilde{\mathbf{w}}_0^T, \tilde{\mathbf{w}}_{n_1}^T, \ldots, \tilde{\mathbf{w}}_{n_N}^T, \tilde{\mathbf{w}}_{l_1}^T, \ldots, \tilde{\mathbf{w}}_{l_L}^T, \tilde{\mathbf{v}}_{t_1}^T, \ldots, \tilde{\mathbf{v}}_{t_T}^T]^T \tag{4.24}$$

This decomposition will change depending on which partitions we use.

As described in the previous section, we can also use mixed l_1 and l_2 norms to automatically select groups or partitions. To select partitions, we combine all the node-related weight vectors $\tilde{\mathbf{w}}_{n_1}, \ldots, \tilde{\mathbf{w}}_{n_N}$ into one long vector $\tilde{\mathbf{w}}_N$, all workload vectors $\tilde{\mathbf{w}}_{l_1}, \ldots, \tilde{\mathbf{w}}_{l_L}$ into $\tilde{\mathbf{w}}_L$ and all $\tilde{\mathbf{v}}_1, \ldots, \tilde{\mathbf{v}}_T$ into $\tilde{\mathbf{v}}_T$. We then solve the following optimization:

$$\boxed{\begin{aligned} \min_{\tilde{\mathbf{w}},b} & \left(\|\tilde{\mathbf{w}}_0\|_2 + \|\tilde{\mathbf{w}}_N\|_2 + \|\tilde{\mathbf{w}}_L\|_2 + \|\tilde{\mathbf{v}}_T\|_2\right)^2 \\ & +C\sum_{i,t} L_{Hinge}\left(y_{it}, \tilde{\mathbf{w}}^T\phi\left(\mathbf{x}_{it}\right)+b\right) \end{aligned} \qquad (4.25)}$$

This model will learn which notion of grouping is most important. For instance if $\tilde{\mathbf{w}}_L$ is set to 0, then we might conclude that straggler behaviour doesn't depend that much on the particular workload being executed.

To select individual groups, we solve the optimization problem:

$$\boxed{\begin{aligned} \min_{\tilde{\mathbf{w}},b,\xi\geq 0} & \left(\|\tilde{\mathbf{w}}_0\|_2 + \sum_{n=1}^{N}\|\tilde{\mathbf{w}}_n\|_2 + \sum_{l=1}^{L}\|\tilde{\mathbf{w}}_l\|_2 + \sum_{t=1}^{T}\|\tilde{\mathbf{v}}_t\|_2\right)^2 \\ & +C\sum_{i,t} L_{Hinge}\left(y_{it}, \tilde{\mathbf{w}}^T\phi\left(\mathbf{x}_{it}\right)+b\right) \end{aligned} \qquad (4.26)}$$

This formulation can set some individual node or workload models to 0. This would mean that straggler behavior on some nodes or workloads can be predicted by generic models, but others require more node-specific or workload-specific reasoning.

We can also use mixed norms for feature selection. The features in our feature vector correspond to resource usage counters, and we divide them into 5 categories, as explained in Section 3.4.1.2: counters based on cpu, those based on network, those based on disk, those based on memory, and other system-level counters. Then each $\tilde{\mathbf{w}}_n$, $\tilde{\mathbf{w}}_l$ and $\tilde{\mathbf{v}}_t$ gets similarly split up. As described before, we can either let each model choose its own set of features using this regularizer:

$$\boxed{\begin{aligned} \Omega(\tilde{\mathbf{w}}) \quad = & \left(\|\tilde{\mathbf{w}}_0^{mem}\|_2 + \|\tilde{\mathbf{w}}_0^{disk}\|_2 + \|\tilde{\mathbf{w}}_0^{cpu}\|_2 + \|\tilde{\mathbf{w}}_0^{network}\|_2 + \|\tilde{\mathbf{w}}_0^{system}\|_2 + \right. \\ & + \textstyle\sum_{n=1}^{N}[\|\tilde{\mathbf{w}}_n^{mem}\|_2 + \|\tilde{\mathbf{w}}_n^{disk}\|_2 + \|\tilde{\mathbf{w}}_n^{cpu}\|_2 + \|\tilde{\mathbf{w}}_n^{network}\|_2 + \|\tilde{\mathbf{w}}_n^{system}\|_2] \\ & + \textstyle\sum_{l=1}^{L}[\|\tilde{\mathbf{w}}_l^{mem}\|_2 + \|\tilde{\mathbf{w}}_l^{disk}\|_2 + \|\tilde{\mathbf{w}}_l^{cpu}\|_2 + \|\tilde{\mathbf{w}}_l^{network}\|_2 + \|\tilde{\mathbf{w}}_l^{system}\|_2] \\ & \left. + \textstyle\sum_{t=1}^{T}[\|\tilde{\mathbf{v}}_t^{mem}\|_2 + \|\tilde{\mathbf{v}}_t^{disk}\|_2 + \|\tilde{\mathbf{v}}_t^{cpu}\|_2 + \|\tilde{\mathbf{v}}_t^{network}\|_2 + \|\tilde{\mathbf{v}}_t^{system}\|_2]\right)^2 \end{aligned} \quad (4.27)}$$

or choose a single set of features globally using this regularizer:

$$\boxed{\Omega(\tilde{\mathbf{w}}) = \left(\|\tilde{\mathbf{w}}^{mem}\|_2 + \|\tilde{\mathbf{w}}^{cpu}\|_2 + \|\tilde{\mathbf{w}}^{disk}\|_2 + \|\tilde{\mathbf{w}}^{network}\|_2 + \|\tilde{\mathbf{w}}^{system}\|_2\right)^2 \qquad (4.28)}$$

where $\tilde{\mathbf{w}}^{mem}$ concatenates all memory related weights from all models,

$$\tilde{\mathbf{w}}^{mem} = [\tilde{\mathbf{w}}_0^{memT}, \tilde{\mathbf{w}}_{n_1}^{memT}, \ldots, \tilde{\mathbf{w}}_{n_N}^{memT}, \tilde{\mathbf{w}}_{l_1}^{memT}, \ldots, \tilde{\mathbf{w}}_{l_L}^{memT}]^T \qquad (4.29)$$

$\tilde{\mathbf{w}}^{cpu}$, $\tilde{\mathbf{w}}^{disk}$ etc. are defined similarly.

4.2.7 Exploring the relationships between the weight vectors

Before getting into the experiments, we can get some insights on what our formulation will learn by looking at the Karush-Kuhn-Tucker (KKT) [34] conditions.

Equation 4.23 can equivalently be written as:

$$\begin{aligned}\min_{\mathbf{w},b,\xi} \lambda_0\|\mathbf{w}_0\|^2 + \frac{\nu}{N}\sum_{n=1}^{N}\|\mathbf{w}_n\|^2 + \frac{\omega}{L}\sum_{l=1}^{L}\|\mathbf{w}_l\|^2 \\ +\frac{\tau}{T}\sum_{t=1}^{T}\|\mathbf{v}_t\|^2 + \sum_{t=1}^{T}\sum_{i=1}^{k_t}\xi_{it} \\ \text{s.t. } y_{it}\left((\mathbf{w}_0 + \mathbf{w}_{n_t} + \mathbf{w}_{l_t} + \mathbf{v}_t)^T\mathbf{x}_{it} + b\right) \geq 1 - \xi_{it} \;\; \forall i,t \\ \xi_{it} \geq 0 \;\; \forall i,t\end{aligned} \tag{4.30}$$

The Lagrangian of the formulation in Equation 4.30 is:

$$\begin{aligned}\mathcal{L}(\mathbf{w},b,\boldsymbol{\alpha},\boldsymbol{\gamma}) = \lambda_0\|\mathbf{w}_0\|^2 + \frac{\nu}{N}\sum_{n=1}^{N}\|\mathbf{w}_n\|^2 + \frac{\omega}{L}\sum_{l=1}^{L}\|\mathbf{w}_l\|^2 \\ +\frac{\tau}{T}\sum_{t=1}^{T}\|\mathbf{v}_t\|^2 + \sum_{t=1}^{T}\sum_{i=1}^{k_t}\xi_{it} - \sum_{t=1}^{T}\sum_{i=1}^{k_t}\gamma_{it}\xi_{it} \\ +\sum_{t=1}^{T}\sum_{i=1}^{k_t}\alpha_{it}\left(1 - \xi_{it} - y_{it}\left(\mathbf{w}_t^T\mathbf{x}_{it} + b\right)\right)\end{aligned} \tag{4.31}$$

Taking derivatives w.r.t. the primal variables and setting to 0 gives us relationships between $\mathbf{w}_0, \mathbf{v}_t, \mathbf{w}_n$ and $\mathbf{w}_l$:

$$\lambda_0\mathbf{w}_0^* = \frac{\tau}{T}\sum_t \mathbf{v}_t^* \tag{4.32}$$

$$\nu\mathbf{w}_n^* = \frac{\tau}{T/N}\sum_{t:n_t=n}\mathbf{v}_t^* \tag{4.33}$$

$$\omega\mathbf{w}_l^* = \frac{\tau}{T/L}\sum_{t:l_t=l}\mathbf{v}_t^* \tag{4.34}$$

$$\lambda_0\mathbf{w}_0^* = \frac{\nu}{N}\sum_n \mathbf{w}_n^* \tag{4.35}$$

$$\lambda_0\mathbf{w}_0^* = \frac{\omega}{L}\sum_l \mathbf{w}_l^* \tag{4.36}$$

[61] also obtain Equation 4.32 in their formulation, but the other relationships are specific to ours. These relationships imply that these variables shouldn't be considered independent. $\mathbf{w}_n$, $\mathbf{w}_l$ and $\mathbf{w}_0$ are scaled means of the $\mathbf{v}_t$'s of the group they capture.

4.2.8 Generalizing to unseen nodes and workloads

Consider what happens when we remove the partition corresponding to individual $\{$node, workload$\}$ tuples, i.e., $\mathbf{v}_t$, from our formulations. We now do not have any parameters specific to a node-workload combination, but can still capture both node- and workload-dependent properties of the learning problem. Such formulations are thus similar to factorized models where the node and workload dependent factors are grouped into separate blocks. We end up with only $(N + L)d$ parameters, whereas a formulation like that of [61, 62, 93] will still have NLd parameters (here d is the input dimensionality). Thus, we can *reduce* the number of parameters while still capturing the essential properties of the learning problems.

In addition, since we no longer have a separate weight vector for each $\{$node, workload$\}$ tuple, we can generalize to node-workload pairs that are completely unseen at train time: the classifier for such an unseen combination t will simply be $\mathbf{w}_0 + \mathbf{w}_{n_t} + \mathbf{w}_{l_t}$. We thus explicitly use knowledge gleaned from prior workloads run on this node (through $\mathbf{w}_{n_t}$) and other nodes running this workload (through $\mathbf{w}_{l_t}$). This is especially useful in our application where there may be a large number of nodes and workloads. In such cases, collecting data for each node-workload pair will be time consuming, and generalizing to unseen combinations will be a significant advantage.

In most of our experiments, therefore, we remove the partition corresponding to individual $\{$node, workload$\}$ tuples. We explicitly evaluate how well we generalize by doing so in Section 4.3.4.

4.3 Empirical Evaluation

In this section, we describe our dataset (Section 4.3.1), provide variants to our proposed formulation that capture node and workload properties (Section 4.3.2). These formulations allow us to gain insight into which subsets of features matter the most in predicting straggler behavior on underlying nodes. We then thoroughly evaluate them using the following metrics: first, classification accuracy when there is sufficient data (Section 4.3.3), and also when sufficient data is not available (Section 4.3.4), and second, improvement in overall job completion times (Section 4.3.5), and third, reduction in resources consumed (Section 4.3.6).

4.3.1 Datasets

The set of real-world workloads considered for evaluating our multi-task learning formulations are the same as those used for Wrangler's evaluation. We repeat the dataset details here for readability. These are collected from the production compute clusters at Facebook and Cloudera's customers, which we denote as $FB2009$, $FB2010$, CC_b and CC_e. Table 4.1 provides details about these workloads in terms of the number of machines in the actual clusters, the length and date of data

Trace	#Machines	Length	Date	#Jobs
FB2009	600	6 month	2009	11,29,193
FB2010	3,000	1.5 months	2010	11,69,184
CC_b	300	9 days	2011	22,974
CC_e	100	9 days	2011	10,790
Total	$\approx$ 4,000	$\approx$ 8.5 months	-	23,32,141

Table 4.1: Dataset. *FB*: Facebook, *CC*: Cloudera Customer.

Workload	No of tasks (Training+Validation)	No of tasks Test
$FB2009$	4,885	13,632
$FB2010$	3,843	38,158
CC_b	5,991	30,203
CC_e	39,014	94,550

Table 4.2: Number of tasks we use for each workload in the train+val and test sets.

capture, total number of jobs in those workloads. Chen, et al., explain the data in further details in [43]. Together, the dataset consists of traces from over about 4000 machines captured over almost eight months. For faithfully replaying these real-world production traces on our 20 node EC2 cluster, we used a statistical workload replay tool, SWIM [44] that synthesizes a workload with representative job submission rates and patterns, shuffle/input data size and output/shuffle data ratios (see [44] for details of replay methodology). SWIM scales the workload to the number of nodes in the experimental cluster.

For each workload, we need data with ground-truth labels for training and validating our models. We collect this data by running tasks from the workload as described above and recording the resource usage counters x_i at a node at the time a task i is launched, and the ground truth label y_i by checking if it ends up becoming a straggler. Then we divide this dataset temporally into a training set and a validation set. In other words, the first few tasks that were executed form the train set and the rest of the tasks form the validation set. In the experiments below, we vary the percentage of data that is used for training, and compute the prediction accuracy on the validation set. We train our final model using two-thirds of this dataset and proceed to evaluate it on our ultimate metric, i.e., job completion times.

To measure job completion times, we then incorporate the trained models into the job scheduler (as in Wrangler). We then run the replay for the workload again, but with a fresh set of tasks. These fresh set of tasks form our test set, and this test set is only used to measure job completion times. Table 4.2 shows the sizes of the datasets.

Each data point is represented by a 107 dimensional feature vector comprising the node's resource usage counters at the time of launching a task on it. We optimize all our formulations using Liblinear [63] for the l_2 regularized variants and using the algorithm proposed by [2] (modified to work in the primal) for the mixed norm variants.

Below, we describe (1) how we use different MTL formulations and prediction accuracy achieved by these formulations, (2) how we learn a classifier for previously unseen node and/or workload and the prediction accuracy it achieves, (3) the improvement in overall job completion times achieved by our formulation and Wrangler over speculative execution, and (4) reduction in resources consumed using our formulation compared to Wrangler.

4.3.2 Variants of proposed formulation

We consider several variants of the general formulation described in Section 4.2. Using a simple squared l_2 regularizer, we first consider $\mathbf{w}_0$, $\mathbf{w}_n$ and $\mathbf{w}_l$, individually:

- f_0: In this formulation, we consider only the global partition in which all learning problems belong to a single group. This corresponds to removing $\mathbf{v}_t$, $\mathbf{w}_n$ and $\mathbf{w}_l$. This formulation thus learns a single global weight vector, $\mathbf{w}_0$, for all the nodes and all the workloads.
- f_n: Here we consider only the partition based on nodes. This corresponds to only learning a $\mathbf{w}_n$, that is, one model for each node. This model learns to predict stragglers based on a node's resource usage counters across different workloads, but it cannot capture any properties that are specific to a particular workload.
- f_l: Here we consider only the partition based on workloads. This means we only learn $\mathbf{w}_l$, i.e., a workload dependent model across nodes executing a particular workload. This model learns to predict stragglers based on the resource usage pattern caused due to a workload across nodes, but ignores the characteristics of a specific node.

The above three formulations either discard the node information, the workload information, or both. We now consider multi-task variants that capture both, node and workload properties:

- $f_{0,t}$: This is the formulation proposed by [61], and corresponds to using the global partition where all learning-tasks belong to one group, and the partition where each learning-task is its own group. This learns $\mathbf{w}_0$ and $\mathbf{v}_t$. Note that this formulation still has to learn on the order of NLd different parameters, and has to collect enough data to learn a separate weight vector for each $\{$node, workload$\}$ combination.
- $f_{0,t,l}$: This formulation extends the formulation in $f_{0,t}$ by additionally adding the partition based on workloads. It learns $\mathbf{w}_0, \mathbf{w}_l$ and $\mathbf{v}_t$.
- $f_{0,n,l}$: We remove the partition corresponding to individual $\{$node, workload$\}$ tuples, removing $\mathbf{v}_t$ entirely and only learning $\mathbf{w}_0$, $\mathbf{w}_l$ and $\mathbf{w}_n$. As described in Section 4.2.8, this formulation reduces the total number of parameters to $(N + L)d$ and can also generalize to unseen $\{$node, workload$\}$ tuples.

For all these formulations, the hyperparameters λ_0, ν, ω and τ were set to 1 wherever applicable. We found this setting to be close to optimal in our initial cross-validation experiments (see Section 4.6 for additional details).

In addition to these, we also consider sparsity-inducing formulations for automatically selecting partitions or groups or blocks of features, as explained in Sections 4.2.3, 4.2.4, and 4.2.6.

- f_{ps} : We use the global, node-based and workload-based formulations thus removing $\mathbf{v}_t$ entirely as in $f_{0,n,l}$ and use mixed l_1 and l_2 norms to automatically select the useful partitions from among these. This model will learn the notion of grouping that is most important. To select partitions, we combine all the node-related weight vectors $\tilde{\mathbf{w}}_{n_1}, \ldots, \tilde{\mathbf{w}}_{n_N}$ into one long vector $\tilde{\mathbf{w}}_N$ and all workload vectors into $\tilde{\mathbf{w}}_L$ as shown below:

$$\tilde{\mathbf{w}} = [\tilde{\mathbf{w}}_0^T, \underbrace{\tilde{\mathbf{w}}_{n_1}^T, \ldots, \tilde{\mathbf{w}}_{n_N}^T}_{\tilde{\mathbf{w}}_N^T}, \underbrace{\tilde{\mathbf{w}}_{l_1}^T, \ldots, \tilde{\mathbf{w}}_{l_L}^T}_{\tilde{\mathbf{w}}_L^T}]^T \tag{4.37}$$

  Thus our weight vector $\tilde{\mathbf{w}}$ is split up into blocks corresponding to node, workload and global models.

- f_{gs} : This is the formulation where we use mixed l_1 and l_2 norms to automatically select groups within a partition. Again, we only consider the global, node-based and workload-based formulations. This formulation can set individual node or workload models to zero, unlike f_{ps}, that can set a complete partition, i.e. in our case, a combined model of all the nodes or all the workloads to zero. This would mean that predicting straggler behavior on some nodes or workloads does not need reasoning that is specific to those nodes or workloads; instead a generic model would work.

Finally, we also try the following two mixed norms formulations for feature selection. As before, both these formulations remove the partition corresponding to individual {node, workload} tuples, i.e., remove $\mathbf{v}_t$.

- f_{fs1} : As explained in Equation 4.27, we divide our features into five categories corresponding to cpu, memory, disk, network and other system-level counters. Then each weight vector gets similarly split up into these categories. This formulation learns which categories of features are more important for some nodes or workloads than others.

- f_{fs2} : This formulation, as given in Equation 4.28, selects a category of features across all the weight vectors of nodes and workloads. This formulation can learn if stragglers in a cluster are caused due to contention for a specific resource.

4.3.3 Prediction accuracy

In this section, we evaluate the formulations described in the previous section for their straggler prediction accuracy. In the following subsection 4.3.3.1, we evaluate formulations that use l_2 regularizers, viz., f_0, f_n, f_l, $f_{0,n,l}$, $f_{0,t}$, and $f_{0,t,l}$. Then, in Section 4.3.3.2, we evaluate the mixed norm formulations (a) that automatically selects partitions, f_{ps}, (b) that automatically selects groups, f_{gs}, and then the formulations (c) f_{fs1}, and (d) f_{fs2} that can automatically select features. We list our formulations with a brief description in Table 4.3.

Formulation	Description
f_0	uses a single, global weight vector
f_n	uses only node-specific weights
f_l	uses only workload-specific weights
$f_{0,n,l}$	uses global, node- and workload-specific weights
$f_{0,t}$	uses global weights and weights specific to {node, workload} tuples
$f_{0,t,l}$	uses global and workload-specific weights and weights specific to {node, workload} tuples
f_{ps}	selects partitions automatically
f_{gs}	selects groups automatically
f_{fs1}	selects feature-blocks automatically for individual groups
f_{fs2}	selects feature-blocks automatically across all the groups

Table 4.3: Brief description of all our formulations.

4.3.3.1 Formulations with l_2 regularizers

We aim at learning to predict stragglers using as small of an amount of data as feasible, as this means shorter data capture time. Note that stragglers are fewer than non-stragglers, so we oversample from the stragglers' class to represent the two classes equally in both, the training and validation sets[3]. Table 4.4 shows the percentage accuracy of predicting stragglers with varying amount of training data. We observe that:

- With very small amounts of data, all MTL variants outperform Wrangler. In fact, all of f_0 to $f_{0,t,l}$ need only one sixth of the training data to achieve the same or better accuracy.
- It is important to capture both node- and workload-dependent aspects of the problem: $f_{0,n,l}$, $f_{0,t}$ and $f_{0,t,l}$ consistently outperform f_0, f_n and f_l.
- $f_{0,t}$ and $f_{0,t,l}$ perform up to 7 percentage points better than Wrangler with the same amount of training data, with $f_{0,n,l}$ not far behind.

For a better visualization, Figure 4.2 shows the comparison of prediction accuracy of these formulations, in terms of percentage true positives and percentage false positives when 50% of total data is available.

Next, we evaluate and discuss the sparsity-inducing formulations, f_{ps} (Equation 4.25), f_{gs} (Equation 4.26), f_{fs1} (Equation 4.27), and f_{fs2} (Equation 4.28).

4.3.3.2 Formulations with mixed l_1 and l_2 norms

Automatically selecting partitions or groups: In this Section, we evaluate f_{ps} and f_{gs}. Table 4.5 shows the prediction accuracy of these formulations compared to f_0, f_n, f_l and $f_{0,n,l}$. f_{ps} and f_{gs} show comparable prediction accuracy to $f_{0,n,l}$. We also found interesting sparsity patterns in the learnt weight vectors.

[3]An alternative to statistical oversampling would be to use class-sensitive mis-classification penalties.

%Training Data	Wrangler [160]	f_0	f_n	f_l	$f_{0,n,l}$	$f_{0,t}$	$f_{0,t,l}$
1	Insufficient data	66.9	63.5	66.5	65.5	63.7	66.2
2	Insufficient Data	67.1	63.3	67.7	67.5	64.3	67.7
5	Insufficient Data	67.5	68.1	69.1	69.8	69.6	69.1
10	63.9	67.8	70.9	69.4	72.3	73.1	72.9
20	67.2	68.0	72.6	70.1	72.9	74.7	74.8
30	68.5	68.5	73.2	70.3	74.1	75.9	75.8
40	69.7	68.2	73.9	70.5	74.3	76.4	76.4
50	70.1	68.5	74.1	70.4	75.3	77.1	77.2

Table 4.4: Prediction accuracies (in %) of various MTL formulations for straggler prediction with varying amount of training data. See Section 4.3.3.1 for details.

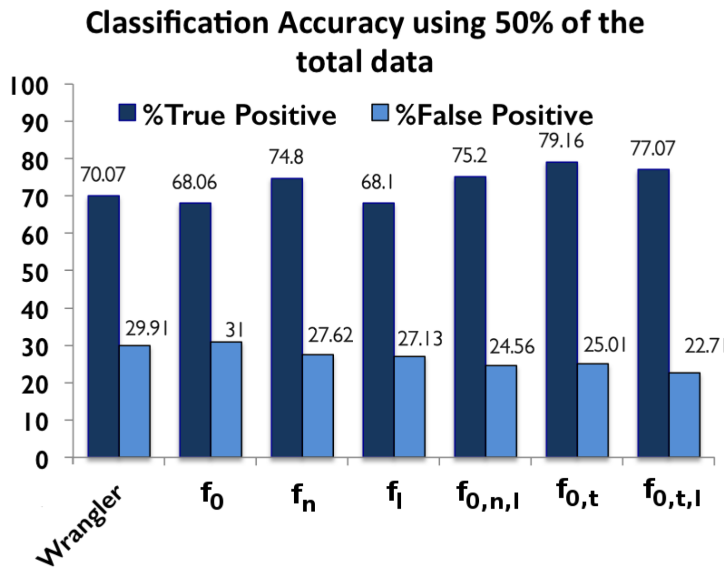


Figure 4.2: Classification accuracy of various MTL formulations as compared to Wrangler using 50% of the total data. This plot shows the percentage of true positives and the percentage of false positives in each of the cases. These quantities are computed as: % True Positive = (fraction of stragglers predicted correctly as stragglers) $\times$ 100, and % False Positive = (fraction of non-stragglers predicted incorrectly to be stragglers) $\times$ 100.

- f_{ps} : Recall that f_{ps} attempts to set the weights of entire partitions to 0. In our case we have a global partition, a node-based partition and a workload-based partition. We observed that only $\tilde{\mathbf{w}}_0$ is zero in the resulting weight vector learned. This means that given node-specific and workload-specific factors, the global factors that are common across all the nodes and workloads do not contribute to the prediction. In other words, similar accuracy could be achieved without using $\tilde{\mathbf{w}}_0$. However, both node- and workload-dependent weights are

Formulation	Straggler Prediction Accuracy
f_0	68.5
f_n	74.1
f_l	70.4
$f_{0,n,l}$	75.3
f_{ps}	74.4
f_{gs}	73.8

Table 4.5: Straggler prediction accuracies (in %) using the four mixed-norm formulations f_{ps} and f_{gs} compared with formulations that use l_2 regularizers. Note that f_{ps} and f_{gs} perform with comparable accuracy with $f_{0,n,l}$, however, use lesser number of groups and parameters, resulting in simpler models.

Formulation	Straggler Prediction Accuracy (in %)
f_{fs1}	74.9
f_{fs2}	74.9

Table 4.6: Straggler prediction accuracies (in %) using f_{fs1} and f_{fs2} that encourage sparsity across bocks of features.

necessary.

- f_{gs} : In f_{gs}, we encourage individual nodes-specific or individual workloads-specific weight vectors separately to be set to zero. We observed that some of the nodes' weight vectors and some of the workload-specific weight vectors were zero, indicating that in some cases we do not need node or workload specific reasoning. (One can use the learnt sparsity pattern and attempt to correlate it with some node and workload characteristics; however we have not explored this in this work.)

We also note that our mixed-norm formulations automatically learn a grouping that achieves comparable accuracy with lesser total number of groups, and thus fewer parameters.

Automatically selecting features: In this section, we evaluate the remaining formulations f_{fs1} and f_{fs2}. These two formulations group sets of features based on resources. We divide the features in five different categories viz., features measuring (1) CPU utilization, (2) memory utilization, (3) network usage, (5) disk utilization, (6) other system level performance counters. We evaluate their straggler prediction accuracy and then discuss their interpretability in terms of understanding the causes behind stragglers.

Table 4.6 shows the percentage prediction accuracies of these formulations on our test set. Note that these formulations show comparable prediction accuracy. Next, we discuss the impact of f_{fs1} and f_{fs2} on understanding the straggler behavior.

- f_{fs1}: Because this formulation divides each group weight vector further into blocks based on the kind of features, it can potentially provide fine-grained insight into what kinds of features are most important for each group weight vector. Indeed, we found that some node models assign zero weight to features from the network category, while others assign a zero weight to

FB2009		FB2010		CC_b		CC_e	
$f_{0,n,l}$	$f_{0,t}$	$f_{0,n,l}$	$f_{0,t}$	$f_{0,n,l}$	$f_{0,t}$	$f_{0,n,l}$	$f_{0,t}$
73.1	45.3	46.7	48.3	50.2	49.4	52.8	68.2
56.2	57.5	57.3	58.7	61.0	53.5	64.4	48.9
63.9	55.5	50.0	48.8	59.4	53.4	48.9	65.1
63.2	47.7	60.6	57.4	55.7	49.5	47.3	73.9
50.7	42.4	51.4	56.2	50.8	44.6	71.2	59.9

Table 4.7: Straggler Prediction accuracies (in %) of $f_{0,n,l}$ and $f_{0,t}$ on test data from an unseen node-workload pair. See Section 4.3.4 for details.

the disk category. However, no global patterns emerge. This reinforces our belief that the causes of stragglers vary quite a bit from node to node or workload to workload.

- f_{fs2}: This formulation considers these feature categories across the various nodes and workloads and provides a way of knowing if there are certain dominating factors causing stragglers in a cluster. However, we observed that none of the feature categories had zero weights in the weight vector learned for our dataset. Again, this means that there is no single, easily discoverable reason for straggler behavior, and provides evidence to the claim made by [11, 12] that the causes behind stragglers are hard to figure out.

4.3.4 Prediction accuracy for a $\{$node, workload$\}$ tuple with insufficient data

One of our goals in this work is to reduce the amount of training data required to get a straggler prediction model up and running. When a new workload begins to execute on a node, we want to learn a model as quickly as possible. Recall that (Section 4.3.3) with enough training data available we found that $f_{0,n,l}$, $f_{0,t}$ and $f_{0,t,l}$ seem to perform similarly, with $f_{0,n,l}$ performing slightly worse. However, as mentioned in Section 4.2.8, formulation $f_{0,n,l}$ has fewer parameters and, because it has no weight vector specific to a particular $\{$node, workload$\}$ tuple, can generalize to new $\{$node, workload$\}$ tuples unseen at train time. This is in contrast to $f_{0,t}$ which has to fall back on $\mathbf{w}_0$ in such a situation, and thus may not generalize as well. In this section, we see if this is indeed true.

We trained classifiers based on $f_{0,n,l}$ and $f_{0,t}$ leaving out 95% of the data of one node-workload pair every time. We then test the models on the left-out data. Table 4.7 shows the percentage classification accuracy from 20 such runs. We note the following:

- For 13 out of 20 classification experiments, $f_{0,n,l}$ performs better than $f_{0,t}$. For 10 out of these 13 cases, the difference in performance is more than 5 percentage points.
- For workloads $FB2009$ and CC_b, we see $f_{0,n,l}$ performs better consistently.
- $f_{0,n,l}$ sometimes performs worse, but in only 3 of these cases is it significantly worse (worse by more than 5 percentage points). All 3 of these instances are in case of the CC_e workload.

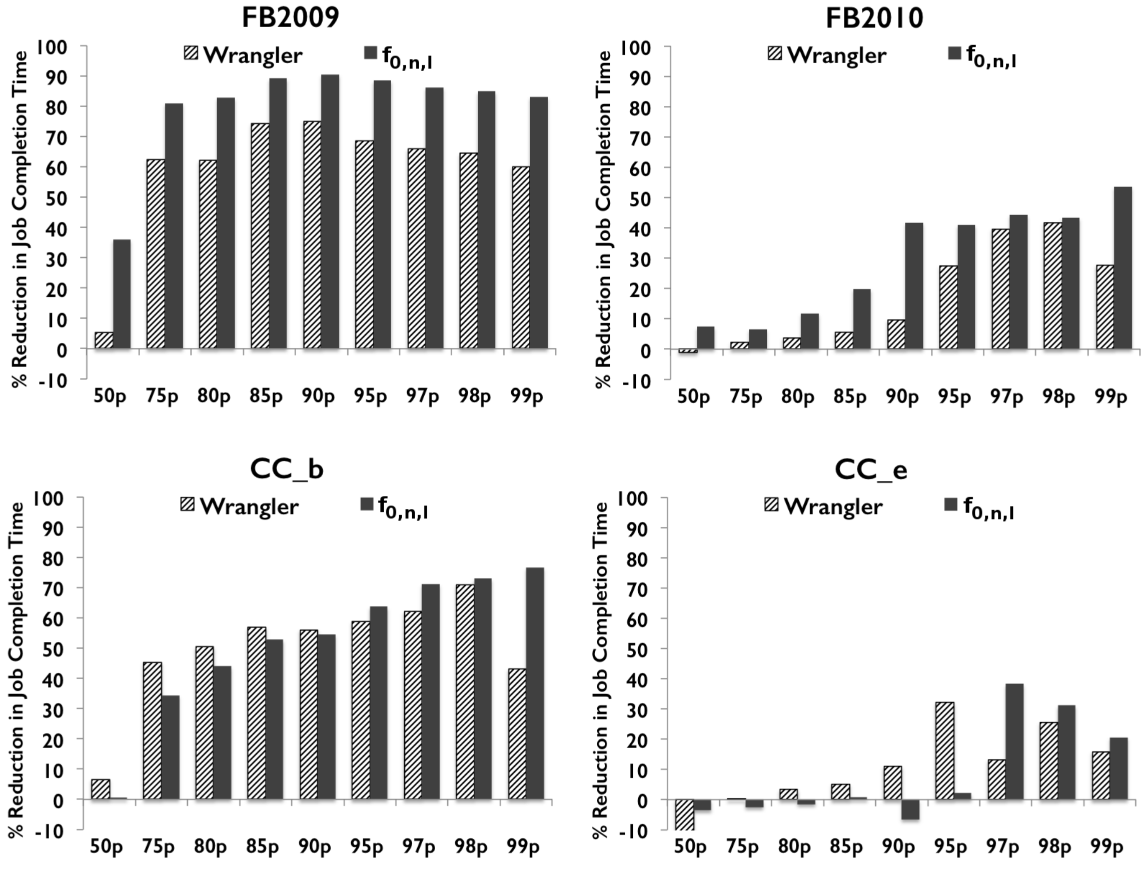


Figure 4.3: Improvement in the overall job completion times achieved by $f_{0,n,l}$ and Wrangler over speculative execution.

In general, for this workload, we also notice a huge variance in the numbers obtained across multiple nodes. See [160], for a discussion of some of the issues in this workload.

This shows that $f_{0,n,l}$ works better in real-world settings where one cannot expect enough data for all node-workload pairs. Therefore, we evaluate $f_{0,n,l}$ in our next experiment (Section 4.3.5) to see if it improves job completion times.

4.3.5 Improvement in overall job completion time

We now evaluate our formulation, $f_{0,n,l}$, using the second metric, improvement in the overall job completion times over speculative execution. We compare these improvements to that achieved by Wrangler (Figure 4.3). Improvement at the 99^{th} percentile is a strong indicator of the effectiveness of straggler mitigation techniques. We see that $f_{0,n,l}$ significantly improves over Wrangler, reflecting the improvements in prediction accuracy. At the 99^{th} percentile, we improve Wrangler's job completion times by $57.8\%, 35.8\%, 58.9\%$ and 5.7% for $FB2009$, $FB2010$, CC_b and CC_e respectively.

Workload	% Reduction in total task-seconds	
	(MTL with $f_{0,n,l}$)	(Wrangler)
FB-2009	73.33	55.09
FB-2010	8.9	24.77
CC_b	64.12	40.15
CC_e	13.04	8.24

Table 4.8: Resource utilization with $f_{0,n,l}$ and with *Wrangler* over speculative execution, in terms of total task execution times (in seconds) across all the jobs. $f_{0,n,l}$ reduces resources consumed over Wrangler for $FB2009$, CC_b and CC_e.

Note that Wrangler is already a strong baseline. Hence, the improvement in job completion times on top of the improvement achieved by Wrangler is significant.

4.3.6 Reduction in resources consumed

When a job is launched on a cluster, it will be broken into small tasks and these tasks will be run in a distributed fashion. Thus, to calculate the resources used, we can sum the resources used by all the tasks. As in [160], we use the time taken by each task as a measure of the resources used by the task. Note that, because these tasks will likely be executing in parallel, the total time taken by the tasks will be much larger than the time taken for the whole job to finish, which is what job completion time measures (shown in Figure 4.3). Ideally, straggler prediction will prevent tasks from becoming stragglers. Fewer stragglers means fewer tasks that need to be replicated by straggler mitigation mechanisms (like speculative execution) and thus lower resource consumption. Thus, improved straggler prediction should also reduce the total task-seconds i.e., resources consumed.

Table 4.8 compares the percentage reduction in resources consumed in terms of total task-seconds achieved by $f_{0,n,l}$ and Wrangler over speculative execution. We see that the improved predictions of $f_{0,n,l}$ reduce resource consumption significantly more than Wrangler for 3 out of 4 workloads, thus supporting our intuitions. In particular, for $FB2009$ and CC_b, $f_{0,n,l}$ reduces Wrangler's resource consumption by about 40%, while for CC_e the reduction is about 5%.

4.4 Related Work on Multi-task Learning

The idea that multiple learning problems might be related and can gain from each other dates back to [145] and [38]. They pointed out that humans do not learn a new task from scratch but instead reuse knowledge gleaned from other learning tasks. This notion was formalized by, among others, [24] and [16], who quantified this gain. Much of this early work relied on neural networks as a means of learning these shared representations. However, contemporary work has also focused on SVMs and kernel machines.

Our work is an extension of the work of [61], who proposed an additive model for MTL that decomposes classifiers into a shared component and a task-specific component. In later work, [62] propose an MTL framework that uses a general quadratic form as a regularizer. They show that if the tasks can be grouped into clusters, they can use a regularizer that encourages all the weight vectors of the group to be closer to each other. [93] extend this formulation when the group structure is not known a priori. [159] infer the group structure using a Bayesian approach. The approach of [156] is similar to ours and groups tasks into "meta-tasks", and tries to automatically figure out the meta-tasks required to get good performance. The formulation we propose is also designed to handle group structure, but allows us to dispense with task-specific classifiers entirely, reducing the number of parameters drastically. This allows us to handle tasks that have very little training data by transferring parameters learnt on other tasks. Our formulation shares this property with that of [27], and indeed our basic formulation can be written down in the kernel-based framework they describe for learning to generalize onto a completely unseen task. Other ways of controlling parameters include learning a distance metric [120], and using low rank regularizers [125].

Our setting is an example of multilinear multitask learning where each learning problem is indexed by two indices: the node and the workload. Previous work on this subfield of multitask learning has typically used low-rank regularizers on the weight matrix represented as tensors [130, 158]. It is also possible to define a similarity between tasks based on how many indices they share [136]. Our formulation captures some of the same intuitions, but has the added advantage of simplicity and ease of implementation.

Using mixed norms for inducing sparsity has a rich history. [59] showed that minimizing the l_1 norm recovers sparse solutions when solving linear systems. When used as a regularizer, the l_1 norm learns sparse models, where most weights are 0. The most well known of such sparse formulations is the lasso [147], which uses the l_1 norm to select features in regression. [165] extend the lasso to group lasso, where they use a mixed l_1 and l_2 norm to select a sparse set of *groups* of features. [22] study the theoretical properties of group lasso. Since these initial papers, mixed norms have found use in a variety of applications. For instance , [126] use a mixed l_∞ and l_1 norm for feature selection. Such mixed norms also show up in the literature on kernel learning, where they are used to select a sparse set of kernels [149, 95] or a sparse set of groups of kernels [2]. [21] provides an accessible review of mixed norms and their optimization, and we direct the interested reader to that article for more details.

4.5 Conclusion

Through this work, we have shown the utility of multitask learning in solving the real-world problem of avoiding stragglers in distributed data processing. We have presented a novel MTL formulation that captures the structure of our learning-tasks and reduces job completion times by up to 59% over prior work [160]. This reduction comes from a 7 percentage point increase in prediction accuracy. Our formulation can achieve better accuracy with only a sixth of the training data and can generalize better than other MTL approaches for learning-tasks with little or no data. We have also presented extensions to our formulation using group sparsity inducing mixed norms that automatically discover the structure of our learning tasks and make the final model more interpretable. Finally, we note that, although we use straggler avoidance as the motivation, our formulation is more generally applicable, especially for other prediction problems in distributed computing frameworks, such as resource allocation [79, 58].

4.6 Additional Details: Cross-validating hyperparameter settings

Our formulation reduces the number of hyperparameters to just one per partitioning, which makes it much easier to cross-validate to set their values. In particular, our formulation in the context of straggler avoidance, Equation (4.23), has four hyperparameters: λ_0, ν, ω, τ. To tune these parameters we used a simple grid search with cross-validation (results are shown in Tables 4.9, 4.10, 4.11, and 4.12). In general we found that the model formulation is relatively robust to the choice of hyperparameters so long as they are within the correct order of magnitude.

λ_0	ν	ω	τ	Accuracy (%)
1	1	1	1	75.38
1	1	1	10	75.57
1	1	1	100	75.24
1	1	1	1000	74.39
1	1	10	1	74.93
1	1	10	10	75.51
1	1	10	100	75.05
1	1	10	1000	74.38
1	1	100	1	74.84
1	1	100	10	74.84
1	1	100	100	73.95
1	1	100	1000	73.04
1	1	1000	1	73.03
1	1	1000	10	72.60
1	1	1000	100	72.11
1	1	1000	1000	71.17
1	10	1	1	74.95
1	10	1	10	75.17
1	10	1	100	74.34
1	10	1	1000	74.05
1	10	10	1	75.42
1	10	10	10	75.05
1	10	10	100	74.69
1	10	10	1000	75.00
1	10	100	1	74.84
1	10	100	10	74.77
1	10	100	100	74.42
1	10	100	1000	73.53
1	10	1000	1	72.88
1	10	1000	10	72.23
1	10	1000	100	72.07
1	10	1000	1000	71.16

λ_0	ν	ω	τ	Accuracy (%)
1	100	1	1	75.39
1	100	1	10	74.95
1	100	1	100	74.42
1	100	1	1000	74.00
1	100	10	1	75.58
1	100	10	10	75.14
1	100	10	100	74.92
1	100	10	1000	73.90
1	100	100	1	74.87
1	100	100	10	74.72
1	100	100	100	74.09
1	100	100	1000	72.95
1	100	1000	1	73.08
1	100	1000	10	72.15
1	100	1000	100	72.47
1	100	1000	1000	70.72
1	1000	1	1	75.47
1	1000	1	10	75.37
1	1000	1	100	74.73
1	1000	1	1000	74.49
1	1000	10	1	75.78
1	1000	10	10	74.73
1	1000	10	100	74.87
1	1000	10	1000	74.40
1	1000	100	1	74.97
1	1000	100	10	75.06
1	1000	100	100	73.68
1	1000	100	1000	72.51
1	1000	1000	1	72.29
1	1000	1000	10	72.02
1	1000	1000	100	71.94
1	1000	1000	1000	70.34

Table 4.9: Tuning the hyperparameters λ_0, ν, ω and τ using grid search.

λ_0	ν	ω	τ	Accuracy (%)
10	1	1	1	65.93
10	1	1	10	67.67
10	1	1	100	63.99
10	1	1	1000	64.67
10	1	10	1	69.97
10	1	10	10	74.92
10	1	10	100	75.13
10	1	10	1000	74.52
10	1	100	1	61.19
10	1	100	10	75.85
10	1	100	100	75.44
10	1	100	1000	75.30
10	1	1000	1	68.35
10	1	1000	10	74.20
10	1	1000	100	74.45
10	1	1000	1000	73.90
10	10	1	1	64.50
10	10	1	10	66.59
10	10	1	100	64.90
10	10	1	1000	61.98
10	10	10	1	69.95
10	10	10	10	75.61
10	10	10	100	75.02
10	10	10	1000	74.63
10	10	100	1	73.26
10	10	100	10	75.19
10	10	100	100	74.77
10	10	100	1000	74.68
10	10	1000	1	58.03
10	10	1000	10	74.83
10	10	1000	100	74.95
10	10	1000	1000	73.83
10	100	1	1	66.44
10	100	1	10	64.54
10	100	1	100	64.98
10	100	1	1000	62.97
10	100	10	1	68.87
10	100	10	10	75.84
10	100	10	100	75.09
10	100	10	1000	75.41
10	100	100	1	69.45
10	100	100	10	75.41
10	100	100	100	75.74
10	100	100	1000	75.30
10	100	1000	1	72.66
10	100	1000	10	75.19
10	100	1000	100	74.30
10	100	1000	1000	74.03
10	1000	1	1	67.33
10	1000	1	10	63.73
10	1000	1	100	61.84
10	1000	1	1000	60.44
10	1000	10	1	67.40
10	1000	10	10	75.60
10	1000	10	100	75.26
10	1000	10	1000	75.30
10	1000	100	1	72.45
10	1000	100	10	75.44
10	1000	100	100	75.30
10	1000	100	1000	75.03
10	1000	1000	1	66.71
10	1000	1000	10	75.00
10	1000	1000	100	75.07
10	1000	1000	1000	74.42

Table 4.10: Tuning the hyperparameters λ_0, ν, ω and τ using grid search.

λ_0	ν	ω	τ	Accuracy (%)
100	1	1	1	67.36
100	1	1	10	66.39
100	1	1	100	63.79
100	1	1	1000	65.24
100	1	10	1	65.46
100	1	10	10	67.84
100	1	10	100	64.63
100	1	10	1000	60.97
100	1	100	1	72.27
100	1	100	10	56.05
100	1	100	100	66.44
100	1	100	1000	70.18
100	1	1000	1	52.14
100	1	1000	10	71.03
100	1	1000	100	65.30
100	1	1000	1000	73.16
100	10	1	1	65.87
100	10	1	10	65.94
100	10	1	100	67.53
100	10	1	1000	64.05
100	10	10	1	69.18
100	10	10	10	62.42
100	10	10	100	62.87
100	10	10	1000	63.63
100	10	100	1	56.48
100	10	100	10	66.43
100	10	100	100	75.18
100	10	100	1000	75.57
100	10	1000	1	53.74
100	10	1000	10	69.14
100	10	1000	100	75.5
100	10	1000	1000	75.24

λ_0	ν	ω	τ	Accuracy (%)
100	100	1	1	64.43
100	100	1	10	65.58
100	100	1	100	65.06
100	100	1	1000	63.42
100	100	10	1	57.59
100	100	10	10	65.14
100	100	10	100	67.08
100	100	10	1000	68.03
100	100	100	1	72.68
100	100	100	10	65.14
100	100	100	100	75.01
100	100	100	1000	75.26
100	100	1000	1	64.70
100	100	1000	10	55.27
100	100	1000	100	75.24
100	100	1000	1000	74.79
100	1000	1	1	65.79
100	1000	1	10	65.15
100	1000	1	100	60.47
100	1000	1	1000	61.25
100	1000	10	1	67.20
100	1000	10	10	63.50
100	1000	10	100	66.72
100	1000	10	1000	61.48
100	1000	100	1	67.65
100	1000	100	10	66.88
100	1000	100	100	75.71
100	1000	100	1000	75.40
100	1000	1000	1	57.90
100	1000	1000	10	69.47
100	1000	1000	100	75.91
100	1000	1000	1000	75.49

Table 4.11: Tuning the hyperparameters λ_0, ν, ω and τ using grid search.

λ_0	ν	ω	τ	Accuracy (%)
1000	1	1	1	64.42
1000	1	1	10	62.57
1000	1	1	100	65.63
1000	1	1	1000	62.00
1000	1	10	1	64.62
1000	1	10	10	65.87
1000	1	10	100	65.36
1000	1	10	1000	65.51
1000	1	100	1	71.64
1000	1	100	10	65.88
1000	1	100	100	61.27
1000	1	100	1000	71.41
1000	1	1000	1	66.75
1000	1	1000	10	74.07
1000	1	1000	100	54.60
1000	1	1000	1000	71.88
1000	10	1	1	63.93
1000	10	1	10	63.35
1000	10	1	100	63.68
1000	10	1	1000	64.02
1000	10	10	1	71.02
1000	10	10	10	65.37
1000	10	10	100	66.78
1000	10	10	1000	64.47
1000	10	100	1	56.41
1000	10	100	10	60.10
1000	10	100	100	68.24
1000	10	100	1000	61.10
1000	10	1000	1	55.50
1000	10	1000	10	66.23
1000	10	1000	100	65.71
1000	10	1000	1000	72.19
1000	100	1	1	63.52
1000	100	1	10	66.93
1000	100	1	100	65.36
1000	100	1	1000	64.37
1000	100	10	1	55.54
1000	100	10	10	62.08
1000	100	10	100	64.24
1000	100	10	1000	62.66
1000	100	100	1	63.72
1000	100	100	10	69.01
1000	100	100	100	61.42
1000	100	100	1000	64.31
1000	100	1000	1	60.21
1000	100	1000	10	71.92
1000	100	1000	100	67.46
1000	100	1000	1000	75.28
1000	1000	1	1	63.09
1000	1000	1	10	68.08
1000	1000	1	100	63.58
1000	1000	1	1000	59.18
1000	1000	10	1	69.81
1000	1000	10	10	68.01
1000	1000	10	100	67.49
1000	1000	10	1000	66.57
1000	1000	100	1	59.24
1000	1000	100	10	61.69
1000	1000	100	100	66.04
1000	1000	100	1000	61.54
1000	1000	1000	1	71.65
1000	1000	1000	10	58.39
1000	1000	1000	100	71.65
1000	1000	1000	1000	75.59

Table 4.12: Tuning the hyperparameters λ_0, ν, ω and τ using grid search.

Chapter 5

Performance-Aware Resource Allocation in the Public Cloud

In the last two chapters, we discussed predictive scheduling to enable improved performance while optimizing utilization of resources that are already provisioned. In this chapter, we focus our attention on the problem of provisioning the right amount of resources in the first place.

As companies of all sizes migrate to cloud environments, increasingly diverse workloads are being run in the Cloud — each with different performance requirements and cost trade-offs [129]. Recognizing this diversity, cloud providers offer a wide range of Virtual Machine (VM) types. For instance, at the time of writing, Amazon [6], Google [76], and Azure [102] offered a combined total of over 100 instance types with varying system and network configurations.

In this chapter, we address the fundamental problem of *accurately* and *economically* choosing the *best VM* for a given *workload* and *user goals*. This choice is critical because of its impact on performance metrics such as runtime, latency, throughput, cost, and availability. Yet determining or even defining the *"best"* VM depends heavily on the users' goals which may involve diverse, application-specific performance metrics, and span tradeoffs between price and performance objectives.

For example, Figure 5.1 plots the runtimes and resulting costs of running a video encoding task on several AWS VM types. A typical user wanting to deploy a workload might choose the cheapest VM type (`m1.large`) and paradoxically end up not just with poor performance but also high total costs. Alternatively, overprovisioning by picking the most expensive VM type (`m2.4xlarge`) might only offer marginally better runtimes than much cheaper alternatives like `c3.2xlarge`. Thus, to choose the right VM for her performance goals and budget, the user needs accurate performance estimates.

Recent attempts to help users select VM types have either focused on optimization techniques to efficiently search for the best performing VM type [4], or extensive experimental evaluation to model the performance cost trade-off [152]. Simply optimizing for the best VM type for a particular goal (as in CherryPick [4]) assumes that this goal is *fixed*; however, different users might prefer different points along the performance-cost trade-off curve. For example, a user might be willing to

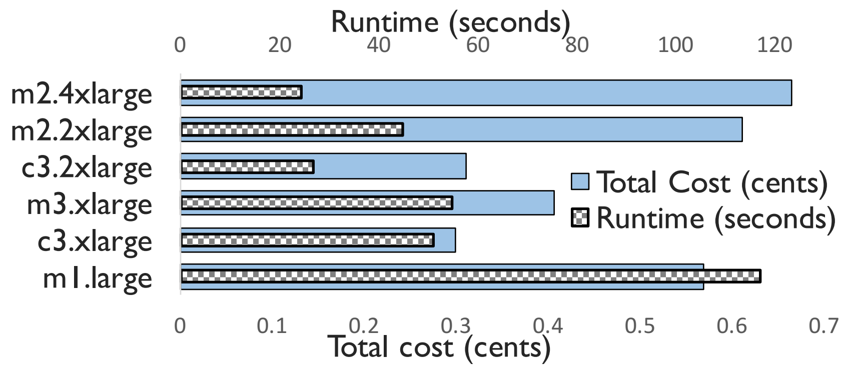


Figure 5.1: Execution time and total cost of a video encoding task on AWS, across various VM types.

tolerate mild reductions in performance for substantial cost savings. In such cases, the user might want to know precisely how switching to another VM type affects performance and cost.

The alternative, directly modeling the performance-cost trade-off, can be challenging. The published VM characteristics (e.g., memory and virtual cores) have hard-to-predict performance implications for any given workload [157, 94, 49]. Furthermore, the performance often depends on workload characteristics that are difficult to specify [58, 25, 94]. Finally, variability in the choice of host hardware, placement policies, and resource contention [129] can result in performance variability [117, 64, 82] that is not captured in the published VM configurations. Recent data driven approaches like Ernest [152] overcome these limitations through extensive performance measurement and modeling. However these techniques introduce an $O(n^2)$ data collection process as each workload is evaluated on each VM type.

The movement towards server-less compute frameworks such as AWS Lambda [19], Azure Functions [20], or Google Cloud Functions [75] may appear to eliminate the challenges of VM selection, but in fact simply shift the challenges to the cloud provider. While cloud providers may have detailed information about their resources, they have limited visibility into the requirements of each workload.

In this chapter, we present PARIS, a **P**erformance**-A**ware **R**esource **I**nference **S**ystem, which estimates the performance-cost trade-off for all VM types, allowing users to balance performance gains with cost reductions. PARIS is applicable to a broad range of workloads and performance metrics and works across cloud providers. PARIS introduces a novel *hybrid* offline and online data collection and modeling framework which provides *accurate* performance estimates with *minimal* data collection, eliminating the $O(n^2)$ data collection complexity.

The key insight in PARIS is to decouple VM performance characterization from the characterization of workload-specific resource requirements. By leveraging a shared profiling framework and established machine learning techniques PARIS is able to combine these separate stages to achieve accurate performance predictions for all combinations of workload and VM type.

In the offline stage, PARIS runs a broad set of benchmarks with diverse resource requirements and collects extensive profiling information for each VM type. Intuitively, the diversity of the resource requirements in the benchmarks ensures that we observe how each VM type responds to demands on its resources. Because these benchmarks are *independent* of the query workloads, the benchmarks only need to be run once for each new VM type.

In the online stage, PARIS characterizes each new query workload by executing a user-specified task that is representative of her workload on a pair of *reference VMs* and collecting the *same* profiling statistics as in the offline stage. These profiling statistics form a *fingerprint* characterizing the workload in the same dimensions as the offline benchmarking process. PARIS then combines this fingerprint with the offline VM benchmarking data to build an accurate model of workload performance across *all* VM types spanning multiple cloud providers.

We demonstrate that PARIS is sufficiently general to accurately predict a range of performance metrics and their variability for widely deployed batch processing and serving-style workloads across VMs from multiple public cloud providers. For instance, it reduces the prediction error for the runtime performance metric by a factor of 4 for some workloads on both AWS and Azure. The increased accuracy translates into a 45% reduction in user cost while maintaining performance (runtime).

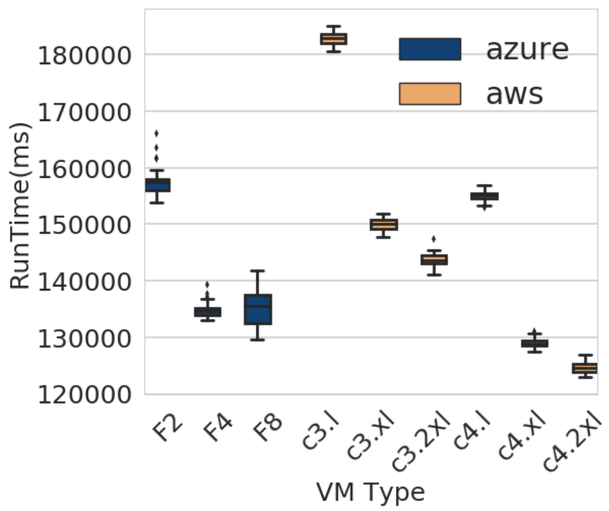
The key contributions of this work are:

- an experimental characterization of performance trade-off of various VM types for realistic workloads across Amazon AWS and Microsoft Azure (Sec. 5.1). We believe to be the first to present this kind of empirical analysis across multiple cloud providers.
- a novel hybrid offline (Sec. 5.3) and online (Sec. 5.4) data collection and modeling framework which eliminates the $O(n^2)$ data collection overhead while providing accurate performance predictions across cloud providers.
- a detailed experimental evaluation demonstrating that PARIS accurately estimates multiple performance metrics and their variabilities (P90 values), for several real-world workloads across two major public cloud providers, thereby reducing user cost by up to 45% relative to strong baseline techniques (Sec. 5.5.3).

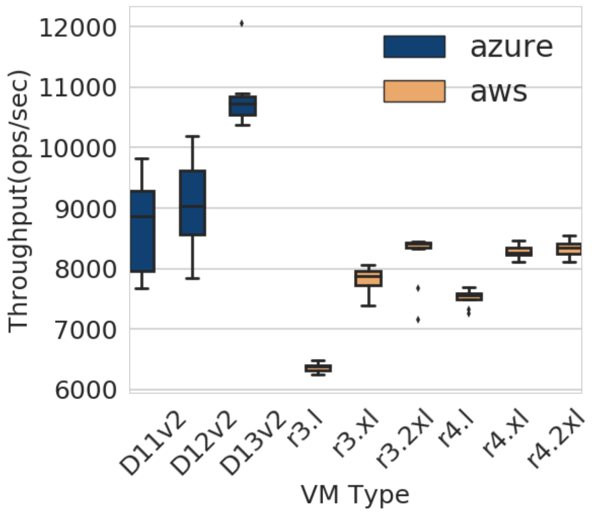
5.1 Insights from Empirical Analysis of Workloads executing across different Public Clouds

To illustrate the challenges involved in selecting VM types, we evaluated three different workloads on a range of VM types spanning two cloud providers: Amazon AWS and Microsoft Azure. Below, we present the complex and often counterintuitive trade-offs between performance and cost that arise when users attempt to understand performance using only the publicly available details of cloud environments. Our observations complement other counterintuitive results observed in the literature [82, 55, 64, 85].

As an example of a software-build system, we studied the compilation of Apache Giraph (see Figure 5.2a) on a range of compute-optimized instances. As an example serving application, we ran a YCSB query processing benchmark on the Redis in-memory data-store (Figure 5.2b) on a



(a) Building Apache Giraph



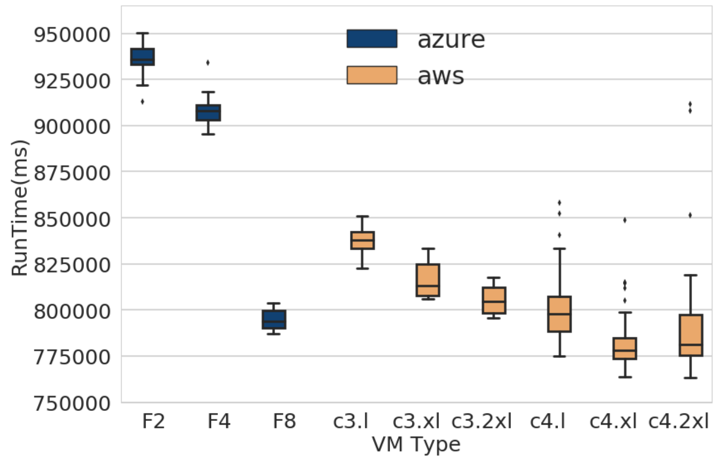
(b) YCSB-benchmarks Workload A

Figure 5.2: **(a)** Runtime for building Apache Giraph (lower the better) and **(b)** Throughput for a 50/50 R/W serving workload on the Redis in-memory datastore using YCSB (higher the better) across different VM types offered by AWS and Azure.

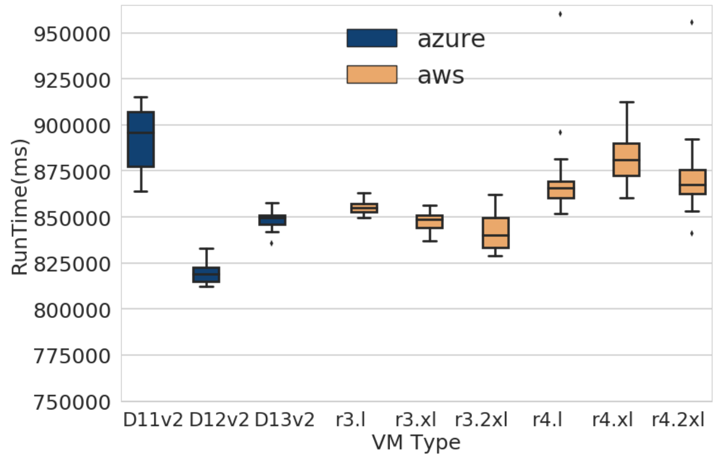
range of memory-optimized instances. Finally, as an example of a more complex task that utilizes multiple resources, we experimented with a compression workload that downloads, decompresses, and then re-compresses a remote file (Figure 5.3). This task emulates many standard cloud-hosted applications, such as video transcoding, that utilize network, compute, disk, and memory at different stages in the computation. We ran the compression workload on specialized as well as general-purpose cloud VMs.

Bigger is not always better. Often users choose to defensively provision the most expensive or the "largest" VM type under the assumption that larger or more expensive instances provide improved performance. This is not always true: for building Giraph, the Azure `F8` VM type performs *worse* than the `F4` VM type in spite of being larger. Similarly, for the YCSB serving benchmark, the throughput does not improve much when going from `r4.xlarge` to the more expensive `r4.2xlarge`, making `r4.xlarge` a more cost-efficient choice. This suggests that provisioning more resources than the workload needs might be unnecessary for good performance.

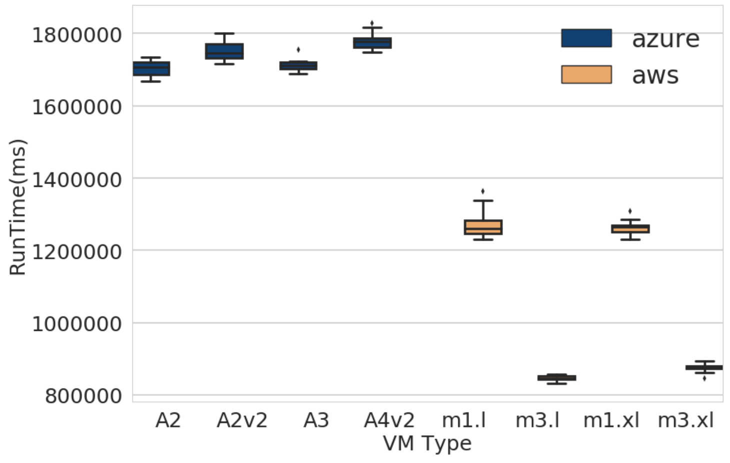
Similar configurations can provide different performance. For the YCSB workload (Figure 5.2b), the AWS `R4` family performs worse than Azure `Dv2` in spite of having a very similar configuration. By contrast, the `R3` and `R4` families perform similarly despite the latter using a newer generation processor. These observations indicate other factors at play: differences in the execution environment, and hardware or software differences that are not reflected in the configuration. Thus, VM type configuration alone does not predict performance.



(a) Compute-optimized VMs



(b) Memory-optimized VMs



(c) General-purpose VMs

Figure 5.3: Runtimes for a compression workload across VM types from different families offered by AWS and Azure. In plot (c), note the different scale on y-axis.

Optimizing for mean performance may not optimize for the tail. For the YCSB workload (Figure 5.2b), Azure VMs provide improved throughput while AWS VMs provide more consistent performance. A developer of a cloud-hosted service might prefer a guaranteed throughput to improved but less predictable throughput. For the compression workload (Figure 5.3), some of the Azure VMs showed reduced variability, even when they lead to a longer expected runtime. Thus, the best VM type may differ depending on whether we are interested in the mean or the tail.

Workload resource requirements are opaque. For workloads that use different resources at different points during their execution, it can be hard to figure out which resources are the most crucial for performance [118]. This is especially challenging for hosted compute services such as AWS-Lambda where the workload is treated as a black-box function. For the compression workload (Figure 5.3), memory- and compute-optimized VM types offered lower runtimes compared to general purpose VM types, indicating that memory or compute, or both, might be the bottleneck. Yet, counterintuitively, going from `r4.l` to `r4.xl`, or `c4.xl` to `c4.2xl` actually *hurts* performance. The processor generation remains the same across the considered VM types. We believe that the counterintuitive performance observed might be because of the underlying execution environment, placement policies, issues of performance isolation, or the non-linear dependence of performance on resource availability. None of these factors is captured in the resource configuration alone.

Monitoring resources consumed while a task is running might help identify resources utilized for that run, but will not tell us how performance is impacted in constrained settings or on different hardware / software. Profiling the workload on each VM across all cloud providers will be informative but prohibitively expensive. Thus, we need a cheaper solution that can nevertheless predict the performance of arbitrary workloads on all VM types accurately.

5.2 PARIS: System Overview

PARIS enables cloud users to make better VM type choices by providing performance and cost estimates on different VM types tailored to their workload.

PARIS runs as a light weight service that presents a simple API to the cloud user. The cloud user (or simply “user”) interacts with PARIS by providing a representative task of her workload, the desired performance metric, and a set of candidate VM types. A representative task is user’s code with a sample input data. The performance metric is either one of a few pre-defined metrics implemented in PARIS, or a snippet of code that implements a function of these metrics. PARIS then predicts the *performance* and *cost* for all of the provided candidate VM types. The user can then use this information to choose the best VM type for any performance and cost goals. For the user, the interaction looks like this:

```
# Get performance and cost est. for targetVMs
perfCostMap = predictPerfCost(userWorkloadDocker, perfMetric,
   candidateVMs)
# Choose VM with min cost subj. to a perf. req.
chosenVMType = minCost(perfCostMap, perfReq)
```

To make accurate performance prediction, PARIS needs to model two things: a) the resource requirements of the workload, and b) the impact of different VM types on workloads with similar resource requirements. However, exhaustively profiling the user's workload on all VM types is prohibitively expensive. To avoid the cost overhead, PARIS divides the modeling task into two phases (Figure 5.4): a *one-time, offline, extensive* VM type benchmarking phase (Section 5.3) and an *online, inexpensive* workload profiling phase (Section 5.4). We provide a high-level overview of each phase below and then elaborate on each phase in the subsequent sections.

In the *offline* VM-benchmarking phase, PARIS uses a *Profiler* to run a suite of benchmarks for each VM type and collect detailed system performance metrics. The benchmark suite is chosen to span a range of realistic *workload patterns* with a variety of resource requirements. This benchmarking can be run by the cloud providers or published[1] by a third party. As new VM types are introduced, the benchmark only needs to be rerun on those new VM types. The offline phase has a fixed one-time cost and removes the extensive profiling and data collection from the critical path of predicting performance characteristics of new user workloads.

In the *online* phase, users provide a representative task and performance metric as described above. PARIS first invokes a *Fingerprint-Generator* that runs the representative task on a *small* (typically, 2) set of *reference VM types* and collects runtime measurements. These measurements capture the resource usage patterns of the task and form the workload *fingerprint*. We choose reference VM types that are farthest apart in terms of their configurations, to capture workload performance in both resource-abundant and resource-constrained settings. Note that while the fingerprinting process incurs additional cost, this cost is small and independent of the number of candidate VM types.

PARIS then combines the fingerprint with the offline benchmarking data to construct a machine learning model that accurately estimates the desired performance metrics as well as the 90^{th} percentile values for corresponding performance metrics for the user workload[2]. Finally, PARIS assembles these estimates into a performance-cost trade-off map across all VM types.

5.3 Offline VM-benchmarking phase

In the offline benchmarking phase, the profiler uses a set of benchmark workloads to characterize VM types. These benchmark workloads are chosen to be diverse in terms of their type, the performance metrics they use, and their resource requirements (Figure 5.5). This allows PARIS to characterize how the different VM types respond to different patterns of resource usage. The set of benchmark workloads is not exhaustive but is intended to span the space of workload requirements. Below we describe the benchmark workloads in more detail.

We used the join and aggregation queries of Hive [146] as representative examples of OLAP-style analytical queries. These model complex analytical queries over structured relational tables and exercise CPU, disk (read), and network. As a representation of latency-sensitive serving workloads in the cloud, we added YCSB core benchmark workloads [50] with Aerospike [1],

[1]We plan to publish our benchmark data across VM types.

[2]PARIS can predict higher percentiles too, but these require more samples during fingerprinting, raising user costs.

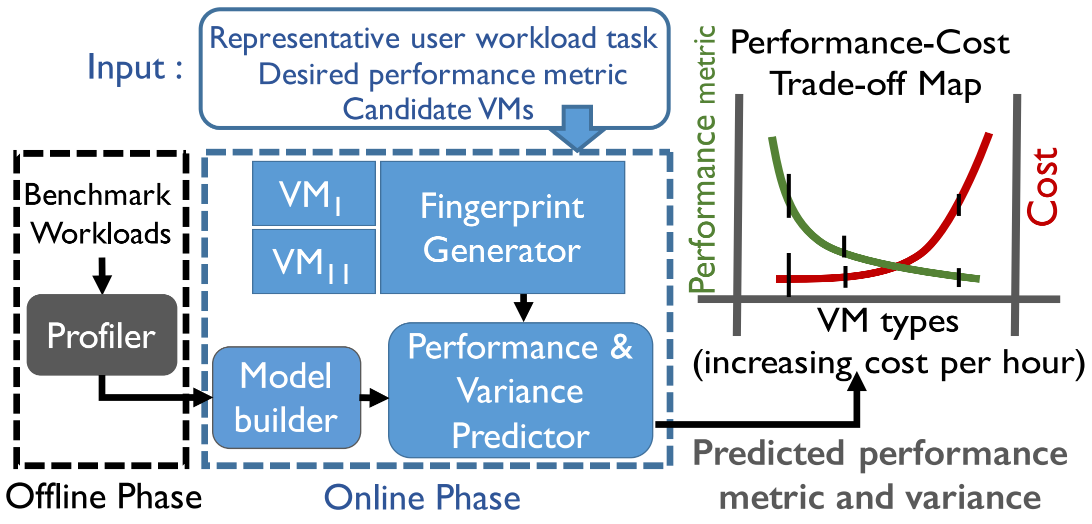

Input :
Representative user workload task
Desired performance metric
Candidate VMs
Benchmark Workloads
Profiler
Offline Phase
VM$_I$
VM$_{II}$
Fingerprint Generator
Model builder
Performance & Variance Predictor
Online Phase
Performance-Cost Trade-off Map
Performance metric
Cost
VM types
(increasing cost per hour)
Predicted performance metric and variance

Figure 5.4: Architecture of PARIS (Sec. 5.2).

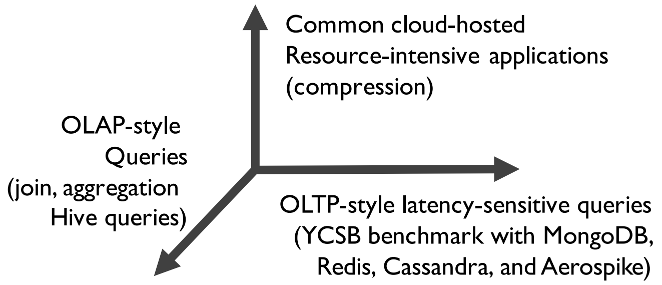

Common cloud-hosted
Resource-intensive applications
(compression)
OLAP-style
Queries
(join, aggregation
Hive queries)
OLTP-style latency-sensitive queries
(YCSB benchmark with MongoDB,
Redis, Cassandra, and Aerospike)

Figure 5.5: Benchmark workloads chosen from a diverse set of cloud use-cases [18].

MongoDB [46], Redis [37], and Cassandra [100] datastores. Finally, as an example of a multi-stage workload, we constructed a benchmark that simulates a hosted compression service, using the squash compression benchmark [139]. This benchmark downloads a compressed file over the network and then decompresses and re-compresses the file thereby exercising compute, memory and disk resources.

The Profiler: The profiler records the performance of each benchmark task for a range of metrics. To estimate performance variability and $p90$ values, each task is run 10 times on each VM type and the empirical 90^{th} percentile performance is computed over all 10 trials. We later explain this in detail in Section 5.5.2. Ten trials is the minimum needed to compute the 90^{th} percentile. This number can be trivially increased, but note that while more trials will improve accuracy, they will also proportionally increase cost.

During each run, the profiler also records aggregated measurements that represent the task's resource usage and performance statistics. This leverages instrumentation mechanisms that are in place in most of today's infrastructure [129]. Concretely, we used Ganglia [107] to instrument the VMs to capture performance and resource counters at a regular 15 second intervals, and record the average (or sum, depending on the counter) of these counters over the task's run. We collected about 20 resource utilization counters spanning the following broad categories:

(a) **CPU utilization:** CPU idle, system, user time, and CPU utilization in the last 1, 5, and 15 minutes.

(b) **Network utilization:** Bytes sent and received.

(c) **Disk utilization:** Ratio of free to total disk space, I/O utilization in the last 1, 5, and 15 minutes

(d) **Memory utilization:** Available virtual, physical, and shared memory, and the cache and buffer space.

(e) **System-level features:** Number of waiting, running, terminated, and blocked threads and the host load in the last 1, 5, and 15 minutes.

5.4 Online performance prediction

As described earlier (Sec. 5.2), in the online phase, users provide PARIS with a representative task from their workload, a performance metric, and a set of candidate VM types.

PARIS first invokes the *Fingerprint-Generator*, which runs the user-specified task on the predefined set of reference VM types[3], and in the process uses the profiler described above to collect resource usage and performance statistics. The user-defined performance metric is used to measure performance. Because we want to predict the 90^{th} percentile performance, we run the task 10 times on each reference VM type and record the 90^{th} percentile performance on these reference VMs. The resource usage measurements, and the mean and 90^{th} percentile performance on the two reference VM types, are put together into a vector F called the *workload fingerprint*. Intuitively, because the fingerprint records resource usage information and not just performance, this fingerprint can help us understand the resource requirements of the task. This can help us predict the workload's performance on other VM types.

The fingerprint tells us the resources used by the task, and the VM type configuration tells us the available resources. For a single task in isolated environments, if the relationship between its performance and the available resources is known, then this information is enough to predict performance. For example, if, when run on a large machine, the profile indicates that the task used 2 GB of memory, and it performs poorly on a reference VM type with 1 GB of memory, then it might perform poorly on other VM types with less than 2 GB of memory. Otherwise, if the task

[3]The reference VM types can also be chosen by the user.

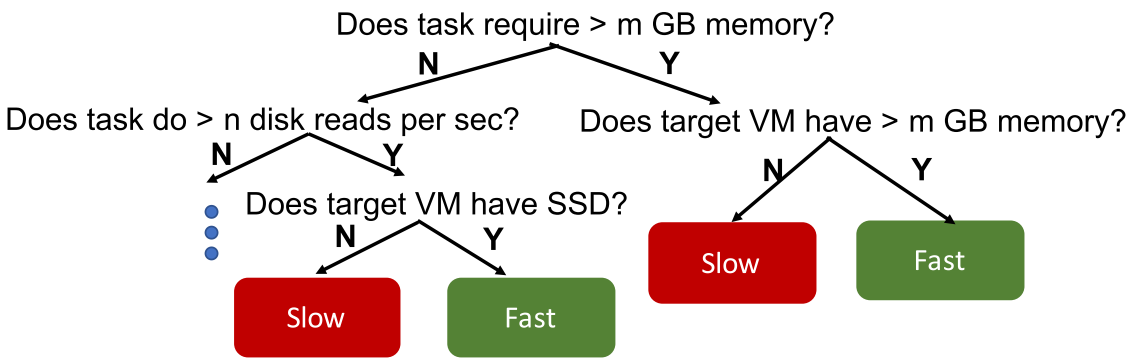

Does task require > m GB memory?
N
Y
Does task do > n disk reads per sec?
N
Y
Does target VM have SSD?
N
Y
Slow
Fast
Does target VM have > m GB memory?
N
Y
Slow
Fast

Figure 5.6: A possible decision tree for predicting performance from the task fingerprint and VM type configuration.

is performing significant disk I/O and spends considerable time blocked on I/O-related system calls, then I/O might be the bottleneck. This kind of non-linear reasoning can be represented as a *decision tree* comprising of a sequence of if-then-else statements (Figure 5.6). Given the workload fingerprint and the candidate (or target) VM configuration, we follow the appropriate path down the tree, finally leading to a performance prediction. Note that a decision tree can make fairly complex, non-linear decisions.

Manually specifying a decision tree for each workload would be prohibitively challenging. We therefore leverage the data collected from the extensive *offline* benchmarking phase in conjunction with established *random forest* algorithms to automatically train a *collection* of decision trees for each workload. Random forests extend the reasoning behind decision trees to a collection of trees to provide more robust predictions [35].

5.4.1 Training the Random Forest Model

To accurately predict the average and tail workload performance using the offline benchmark data we train a random forest model which approximates the function:

$$g(\texttt{fingerprint, target\_vm}) \rightarrow (\texttt{perf, p90})$$

To learn this function we transform the offline benchmarking dataset into a training dataset where each benchmark has a corresponding fingerprint and known mean and tail performance for all target VM types.

The fingerprint for each benchmark task is easily obtained by putting together the resource utilization counters collected while running the task on the reference VMs. Because we profile each benchmark on each VM type in the offline phase, these resource utilization counters are available irrespective of the choice of reference VM types. The target VM in our model is characterized by the VM configuration consisting of the number of cores (Azure) or vcpus (AWS), amount of memory, disk size, and network performance and bandwidth.

Similarly, using the performance statistics collected during the offline benchmarking phase in conjunction with the user-specified performance metric, we compute both mean and tail performance for each benchmark which we use as the targets when training our model. We feed this training dataset to an off-the-shelf random forest training algorithm [121]. In our experiments, training a random forest predictor took less than 2 seconds in most cases. As an implementation detail, instead of predicting absolute performance, we predict the performance scaling relative to the first reference VM type. We found that this led to a simpler learning problem.

5.4.1.1 Invoking the performance predictors

Once the model builder has trained random forests for the performance metric of interest, for each candidate VM type j, we feed the user task fingerprint F and the VM configuration c^j as inputs to the two random forests. The random forests output the mean and 90^{th} percentile performance relative to the first reference VM. We get absolute performance by multiplying these predictions with the corresponding mean and 90^{th} percentile performance on the first VM type.

5.4.1.2 Performance-Cost Map

Finally, PARIS uses the performance predictions to also estimate the cost for each VM type. For this we assume that the cost is a function of the performance metric and the published cost per hour of the VM, that is either known (for standard performance metrics such as throughput or latency) or specified by the user as an additional argument in the call to PARIS. For example, for a serving-style workload where performance is measured by latency, then the total cost per request would be the latency times the published cost per hour.

PARIS' estimated performance-cost trade-off enables users to implement a high-level policy to pick a VM type for a given workload. For example, a policy could be to choose a VM type for a workload that has: (a) an estimated cost below a certain constraint C and (b) the best performance in the worst case. We specify the worst case performance with a high percentile execution time, such as 90^{th} percentile. An alternative policy might pick an "optimal" VM type that achieves the least cost and the highest predictable worst-case performance.

5.4.2 Interpreting the Learned Models

Figure 5.7 illustrates the top 5 features that the random forest considers important, for runtime prediction on AWS and Azure. Here feature importance is based on the intuition that the decision tree will make early splits based on the most informative features, and then gradually refine its prediction using less informative features. Thus important features are those which frequently appear near the top of the decision tree. We find that various measures of CPU usage and the number of CPUs in the target VM figure prominently, for both AWS and Azure. This makes sense, since in general the more CPUs, the more the compute available to the task. However, measures of memory usage and disk utilization are also important. Note that the actual features that are used to estimate performance will depend on the path taken down the tree, which in turn will be different for different workloads.

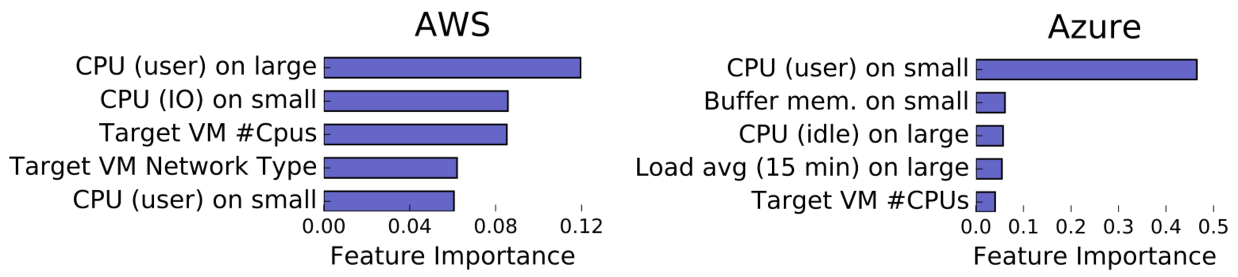


Figure 5.7: Importance of the various features for AWS (left) and Azure (right). The random forests were trained to predict runtime using a compression benchmark workload suite (See Section 5.5.2). Reference VMs used: `c3.large` and `r3.2xlarge` for AWS and `F2` and `D13v2` for Azure.

5.5 Evaluation

In this section we answer following questions:

1. **Prediction accuracy (Section 5.5.3):** How accurately does PARIS predict the mean and 90^{th} percentile values for different performance metrics?
2. **Robustness (Section 5.5.4):** Is PARIS robust to changes in (a) the number and choice of VM types (5.5.4.1, 5.5.4.2), (b) the benchmark workloads used in the offline profiling phase (5.5.4.5), and (c) the choice of modeling technique (regressor) (5.5.4.3, and 5.5.4.4)?
3. **Usefulness (Sections 5.5.5, 5.5.6):** (a) Can we convert PARIS' performance estimates into actionable information (5.5.5) that reduces cost (5.5.6)?

5.5.1 Baselines

No off-the-shelf approach exists for predicting the performance of arbitrary workloads on all VM types in the cloud. Often users defensively provision the most expensive VM type, but this can lead to excessive costs without performance gains (Sec. 5.1). Alternatively, exhaustively profiling the workload on every available VM type provides accurate performance estimates, but is prohibitively expensive.

Instead, we chose baselines described below that are similar to PARIS in terms of user cost, use the published VM configurations intelligently, and correspond to what users might do given the available information and a tight budget:

(a) Collaborative Filtering: Similar to PARIS, Paragon [57] and Quasar [58] also work by profiling key performance metrics and extrapolating performance on different VM types using matrix completion or collaborative filtering [127]. We compare PARIS with a representative collaborative filtering method: The initial dense matrix is generated using PARIS' offline benchmarking data. Benchmark workload tasks form the rows of this matrix, the VM types form the columns, and each cell (i, j) in the matrix contains the value of the performance metric for i-th benchmark workload task on the j-th VM type. When a user-workload task is profiled on two VM types, we append a row with only two values corresponding to those two VMs to this matrix. Using matrix completion, we get the performance prediction on rest of VM types in the matrix for the user workload task.

(b) Baseline$_1$: To reduce the cost, a user might profile her workload on the "smallest" and "largest" VM types according to some resource, and then take *average* performance to be an estimate on an intermediate VM type. Concretely, suppose VM type 1 obtains performance, (for instance, runtime), p_1, and VM type 2 achieves performance p_2. Then for a target VM type, one might simply predict the performance to be $p_{target} = \frac{p_1+p_2}{2}$.

(c) Baseline$_2$: Instead of simply averaging the performance, Baseline$_2$ *interpolates* performance based on published configurations. Concretely, suppose VM type 1 has memory m_1 and gets performance p_1, and VM type 2 has memory m_2 and gets performance p_2. Then for a VM type offering memory m, one might simply predict the performance to be $p_{memory}(m) = p_1 + \frac{p_2-p_1}{m_2-m_1}(m - m_1)$. Since the user may not know which resource is important, she might do such linear interpolation for each resource and average the predictions together.

5.5.2 Experimental Set-up

We evaluated PARIS on AWS and Azure, using two widely recognized types of cloud workloads [9]: (a) Applications such as video encoding, and compression, and (b) Serving-style latency and throughput sensitive OLTP workloads.

Common cloud-hosted applications: Video encoding and compression are common use-cases of the cloud. We used the squash compression library [139], an abstraction layer for different compression algorithms that also has a large set of datasets. For a video encoding workload, we used libav [115], a set of open source audio and video processing tools. We set both of these applications in the cloud, to first download the relevant input data and then process it. The video encoding application first downloads a video using the specified URL, then converts it to a specified format using various frame-rates, codecs, and bit-rates. The compression workload downloads a compressed file, decompresses it, and re-compresses it using different compression algorithms. These workloads have different resource usage patterns. To show that PARIS can generalize well across workloads, we chose the compression application for the offline benchmarking and tested the models using the video encoding application (Table 5.1).

Serving-style workloads: We used four common cloud serving datastores: Aerospike, MongoDB, Redis, and Cassandra. These systems provide read and write access to the data, for tasks like serving a web page or querying a database. For querying these systems, we used multiple workloads from the YCSB framework [50]. We used the core workloads [163], which represent different mixes of read/write operations, request distributions, and datasizes. Table 5.2 shows the benchmark serving workloads we used in the offline phase of PARIS. For testing PARIS' models, we implemented new

Workload	Number of tasks	Time (hours)
Cloud hosted compression (Benchmark set)	740	112
Cloud hosted video encoding (Query set)	12,983	433
Serving-style YCSB workloads D, B, A (Benchmark set)	1,830	2
Serving-style new YCSB workloads (Query set)	62,494	436

Table 5.1: Details of the workloads used and Dataset collected for PARIS' offline (benchmark) and online (query) phases.

Workload	Operations	Example Application
D	Read latest: 95/5 reads/inserts	Status updates
B	Read mostly: 95/5 reads/writes	Photo tagging
A	Update heavy: 50/50 reads/writes	Recording user-actions

Table 5.2: Serving benchmark workloads we used from YCSB. We did not use the Read-Only Workload C, as our benchmark set covers read-mostly and read-latest workloads.

realistic serving workloads by varying the read/write/scan/insert proportions and request distribution, for a larger number of operations than the benchmark workloads [164].

Dataset details: Table 5.1 shows the number of tasks executed in the offline phase (benchmark set) and the corresponding amount of time spent. Also shown are the workloads and the number of query tasks used for online evaluation (query set).

Metrics for evaluating model-predictions: We use the same error metrics for our predictions of different performance metrics. We measured actual performance by running a task on the different VM types as ground truth, and computed the percentage RMSE (Root Mean Squared Error), relative to the actual performance:

$$\%Relative\ RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{p^i - a^i}{a^i}\right)^2} * 100$$

where N is the number of query tasks, and p^i and a^i are the predicted and actual performance of the task respectively, in terms of the user-specified metric. We want the % Relative RMSE to be as low as possible.

RMSE is a standard metric in regression, but is scale-dependent: an RMSE of 10 ms in runtime prediction is very bad if the true runtime is 1 ms, but might be acceptable if the true runtime is 1000 ms. Expressing the error as a percentage of the actual value mitigates this issue.

5.5.3 Prediction accuracy of PARIS

We first evaluate PARIS' prediction accuracy by comparing PARIS' predictions to the actual performance we obtained as ground truth by exhaustively running the same user-provided task on all VM types. We evaluated PARIS on both AWS and Azure for (a) Video encoding tasks using *runtime* as the target performance metric, and (b) serving-type OLTP workloads using *latency* and *throughput* as the performance metrics.

Overall Prediction Error: Figure 5.8 compares PARIS' predictions to those from Baseline$_\mathbf{1}$ and Baseline$_\mathbf{2}$ for the mean and 90^{th} percentile runtime, latency and throughput. Results are averaged across different choices of reference VMs, with standard deviations shown as error bars.

PARIS reduces errors by a factor of 2 compared to Baseline$_1$, and by a factor of 4 compared to Baseline$_2$. Note that the cost of all three approaches is the same, corresponding to running the user task on a few reference VMs. This large reduction is because the nonlinear effects of resource availability on performance (such as hitting a memory wall) cannot be captured by linear interpolation (Baseline$_\mathbf{2}$) or averaging (Baseline$_\mathbf{1}$).

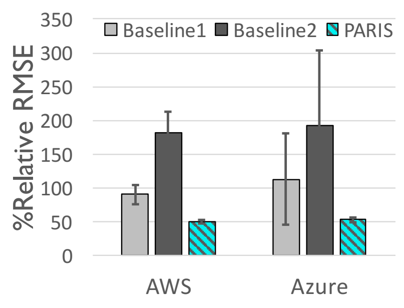
To better understand why Baseline$_\mathbf{2}$ gets such a high error for some VM types, we looked at how predictions by Baseline$_\mathbf{2}$ varied with the different resources of the target VMs (num CPUs, memory, disk). In one case, when using `m3.large` and `c4.2xlarge` as our reference VMs, we observed that surprisingly, Baseline$_\mathbf{2}$ predicted *higher* runtimes for VM types with higher disk capacity. Why did the baseline latch on to this incorrect correlation? In this example, the larger reference VM we used, `c4.2xlarge`, offered lower runtimes than the smaller reference VM used, `m3.large`; however, the smaller reference VM had larger disk (32GB) than the larger reference VM.

This reveals a critical weakness of the baseline: from only the performance on two reference VMs and the published configurations, the baseline *cannot* know which resource is important for workload performance. PARIS, on the other hand, looks at the usage counters and might figure out that disk is not the bottleneck for this workload.

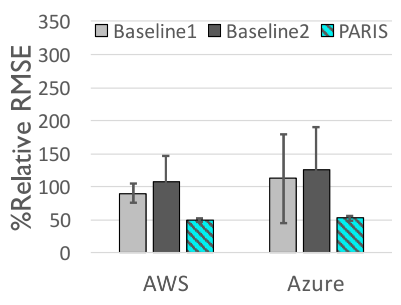
We also note that prediction errors are in general larger for Azure for latency and throughput prediction on the OLTP workloads. We surmise that this is probably due to the higher variability of performance on Azure instances for these workloads, which we pointed out in Section 5.1.

Comparison with Collaborative Filtering Baseline: The prediction errors for mean runtime, mean latency and mean throughput using the collaborative filtering baseline were higher than Baseline$_\mathbf{1}$, Baseline$_\mathbf{2}$, and PARIS. For mean runtime prediction, the % relative RMSE obtained using collaborative filtering was over 543.5%, and for mean latency and mean throughput, it was 80.8% and 43.9% respectively on AWS. This low performance, even relative to the other two baselines, is possibly because matrix completion techniques ignore *both* the VM configuration and the resource utilization information used by PARIS. Given the higher error of this baseline compared to the the other two baselines, we drop it from further evaluation results and discussion.

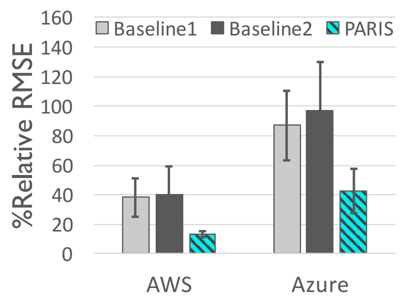
Prediction Error per VM-type: Figure 5.9 shows how the prediction error breaks down over different target VMs. This is a representative result, with mean latency as the performance metric.



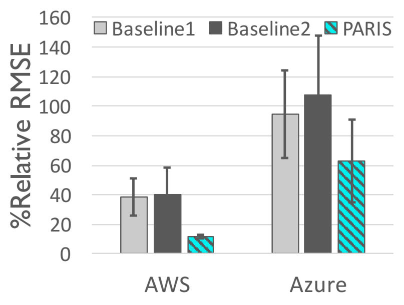
(a) Prediction target: Mean runtime



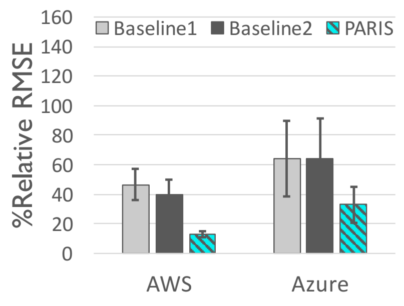
(b) Prediction target: p90 runtime



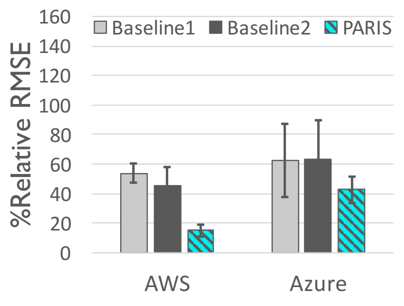
(c) Prediction target: Mean latency



(d) Prediction target: p90 latency



(e) Prediction Mean throughput



(f) Prediction target: p90 throughput

Figure 5.8: Prediction Error for Runtime, Latency, and Throughput (expected and p90) for AWS and Azure. a, b: Runtime prediction for video encoding workload tasks, c-f: Latency and throughput prediction for Serving-style latency and throughput sensitive OLTP workloads. The error bars show the standard deviation across different combinations of reference VMs used.

Reference VM types are `m3.large` and `c4.2xlarge` on AWS, and `A2` and `F8` on Azure. PARIS' errors are *consistently low* across VM types and much lower than both baselines.

As before, Baseline$_2$ discovers spurious correlations and thus often performs worse than Baseline$_1$. Further, *both* baselines perform significantly worse than PARIS for most VM types, perhaps because they lack access to (i) PARIS' offline VM benchmarking data, and (ii) PARIS' resource-utilization statistics, collected in the offline phase as well as when fingerprinting in the online phase.

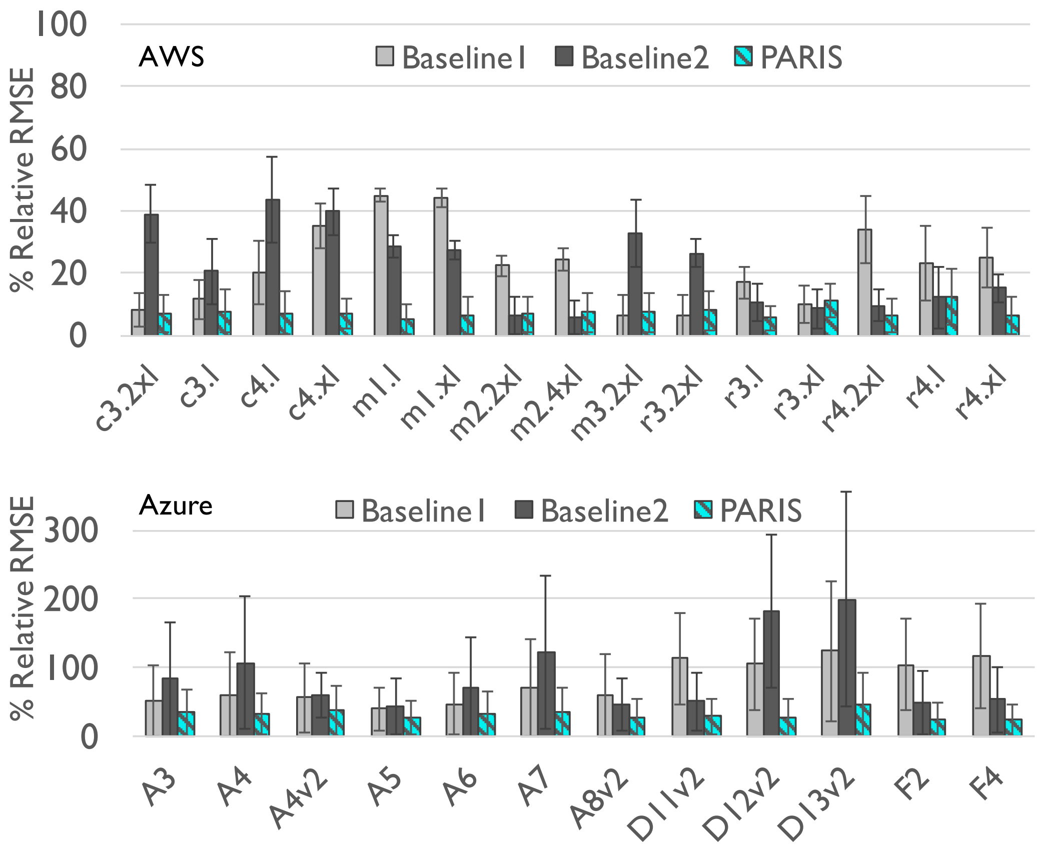


Figure 5.9: Errors per target VM type for predicting mean latency, on AWS (top) and Azure (bottom). Reference VMs used on AWS: `m3.large` and `c4.2xlarge`, and on Azure: `A2` and `F8`. Error bars show the standard deviation in % Relative RMSE across the set of over 62K latency-sensitive YCSB queries run on Aerospike, Cassandra, MongoDb, and Redis data-stores.

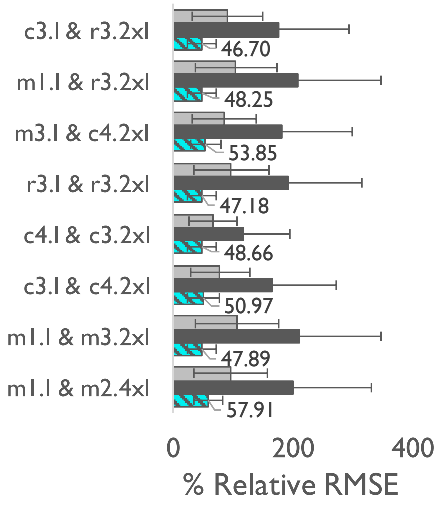
5.5.4 Robustness

PARIS makes a few choices, such as the number and types of reference VMs, modeling algorithms, values of hyperparameters for the performance predictor models, and benchmark workloads. In this section, we evaluate PARIS' sensitivity to these different choices.

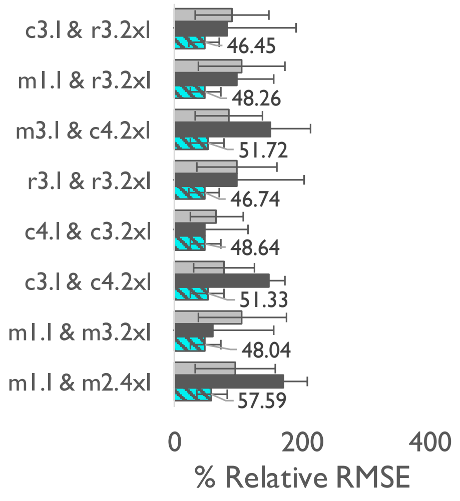
5.5.4.1 Sensitivity to the choice of reference VM types

We experimented with several choices for the 2 reference VM types. We picked pairs of VM types that were the farthest apart in terms of a particular resource (number of cores, amount of memory, disk or storage bandwidth). We also experimented with randomly chosen reference VMs. In general, we found PARIS' predictors to be robust to the choice of reference VM types.

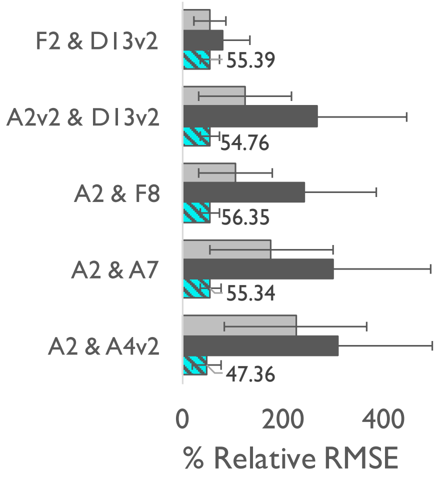
As a representative result, Figure 5.10 compares PARIS' mean and $p90$ runtime predictions to the baselines for several reference VM choices using the video encoding workload. PARIS



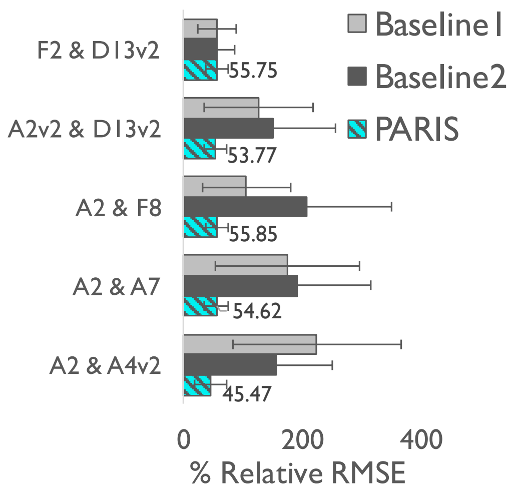
(a) Mean Runtime prediction on AWS



(b) $p90$ runtime prediction on AWS



(c) Mean Runtime prediction on Azure



(d) $p90$ runtime prediction on Azure

Figure 5.10: Sensitivity of PARIS to the choice of Reference VMs. Errors in predicting mean runtime and 90^{th} percentile runtime for video encoding tasks using different reference VM types on AWS (a and b) and Azure (c and d) (Sec. 5.5.4.1).

is both more accurate and more consistent across different reference VM choices. Thus, *PARIS maintains accuracy irrespective of the choice of reference VM types*. The profiling information used by PARIS is consistent and reliable, even when the performance on the two reference VM types is not informative. Further, separate predictors for each selection of reference VM types allow PARIS to learn how performance on the reference VM types extrapolates to other VM types. In contrast, with only the runtime on the two VMs and their published configurations to rely on, the baseline latches on to spurious correlations, and is thus inaccurate.

5.5.4.2 Sensitivity to number of reference VMs

We experimented with increasing the number of reference VMs from 2 (`m3.large` and `c4.2xlarge`) to 3 (`m3.large`, `c4.2xlarge` and `c3.large`) and 4 (`m3.large`, `c4.2xlarge`, `c3.large`, and `c3.xlarge`). We found that latency and throughput prediction error decreased slightly or remained the same as the number of reference VM types increased. % Relative RMSE for latency prediction remained around 9%, while for throughput prediction, it decreased from 11.21% with 2 reference VMs to 10.38% with 3 and 10.27% with 4.

Similarly on Azure, the latency prediction error dropped slightly from 22.89% with 2 reference VMs (`A2` and `A7`) to 21.85% with an additional reference VM (`D13v2`) and to 19.69% with a 4^{th} additional reference VM (`F2`). Throughput prediction error similarly decreased from 24.69% (2 reference VMs) to 18.56% and 18.21% respectively.

This indicates that PARIS is robust to the number of reference VM types and is able to make accurate predictions *with only 2 reference VM types*. This is because of the informative profiling data used by PARIS. To capture how performance varies across VM types, we need to profile using at least two reference VMs.

5.5.4.3 Importance of the choice of regressor

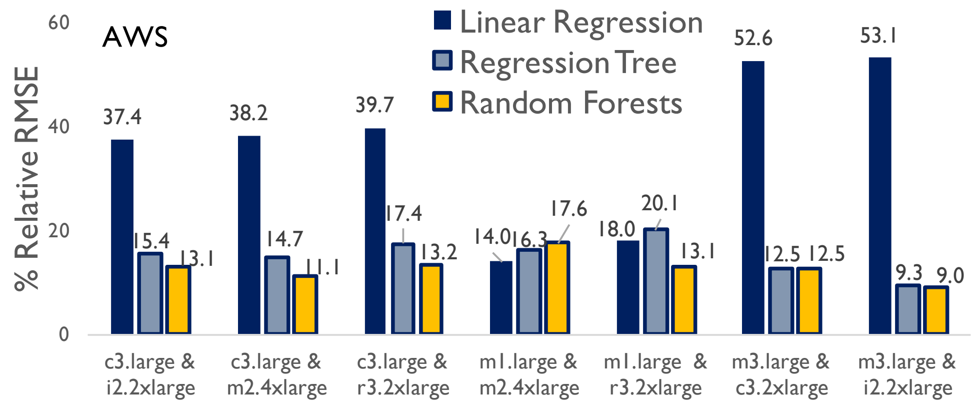
Besides random forests, we also experimented with linear regression and decision trees for throughput and latency prediction on AWS (Figure 5.11). We note that similar patterns emerged using Azure VMs. Linear regression performs the worst as it isn't able to capture non-linear relationships between resource utilization and performance, but owing to the availability of resource usage counters still performs better than **Baseline$_2$**. Regression trees and forests perform equally better, but the forest provides better accuracy by combining complementary trees.

5.5.4.4 Sensitivity to random forest hyperparameters

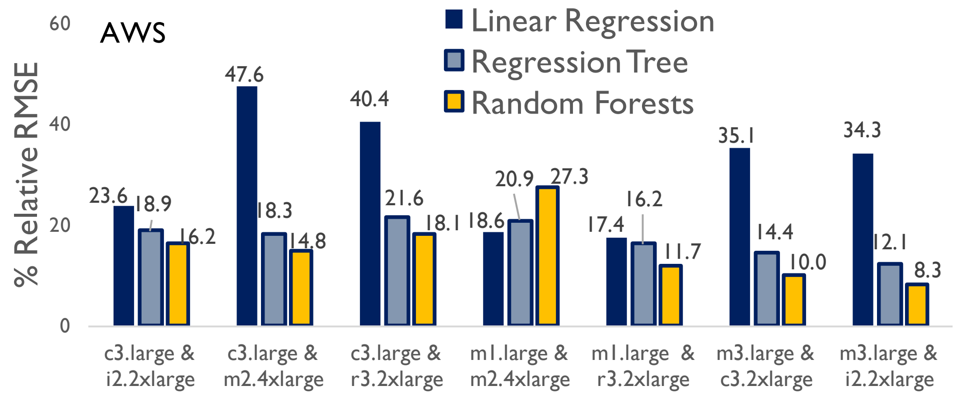
Figure 5.12 shows the percentage relative RMSE of PARIS' latency and throughput predictors for different values of the two most important hyperparameters used by the random forest algorithm: (i) Number of features used per tree (NF), and (ii) Maximum depth of the trees (MD). The predictors for latency and throughput achieve comparable accuracies across the different values of NF and MD. This suggests that *the predictors are robust to hyperparameter choices*.

5.5.4.5 Sensitivity to benchmark workloads

Figure 5.13 shows the percentage relative RMSE of PARIS' latency and throughput predictors when one of the benchmark workloads is removed from the training set at a time. This figure shows the error averaged over different combinations of reference VM types and the error bars indicate the standard deviation. The predictors achieve comparable accuracy on removal of a benchmark workload. We observed a similar trend using the data on Azure for runtime, latency and throughput predictors of PARIS. This shows that *the predictors are robust to different choices of the benchmark workloads*.



(a) Latency prediction



(b) Throughput prediction

Figure 5.11: Prediction errors for different regressors using different choices of reference VM types on AWS (Sec. 5.5.4.3)

5.5.5 From Estimated Performance to Action

PARIS presents its performance predictions as a *performance-cost trade-off map* that maps each VM type to the corresponding performance-cost trade-off, for a given user workload. We first qualitatively explain why we expect this map to be useful and then quantitatively show cost-savings in the next section.

Why do common VM selection strategies fail? Without good estimates of performance or cost, users wanting to deploy workloads on the cloud might:

(a) Try to minimize cost by choosing the cheapest VM.

(b) Defensively choose a large enough VM, assuming 'the higher the cost, the better the performance', or

(c) Pick the largest VM cheaper than a cost constraint.

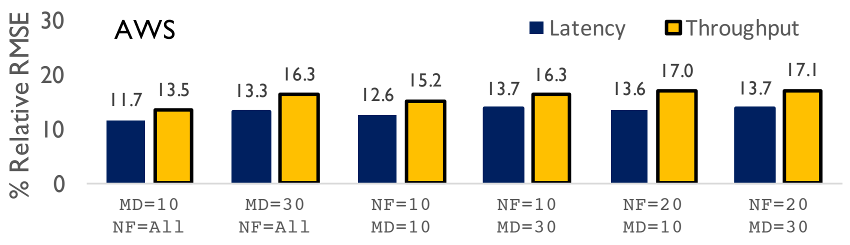


Figure 5.12: Error of PARIS' latency and throughput predictors for different random forest hyperparameters, for test YCSB workloads on Aerospike, MongoDB, Redis, and Cassandra datastores, on AWS. Reference VMs: c3.large & i2.2xlarge. (Sec. 5.5.4.4)

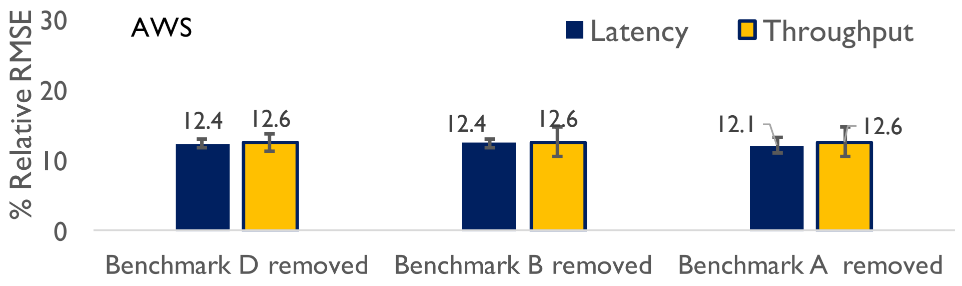


Figure 5.13: Error of PARIS' latency and throughput predictors by removing one benchmark workload at a time, averaged across reference VM types combinations on AWS (Sec. 5.5.4.5).

Figure 5.1 shows the actual performance and cost for a video encoding task on each VM type. Note that this information is unavailable to users unless they are willing to incur heavy profiling-costs. We can see that strategy a) would choose `m1.large`, and lead to higher costs and higher and less predictable runtimes, possibly violating Service Level Objectives (SLOs): a bad decision. Strategy b) would select `m2.4xlarge` and keep runtime low and predictable but incur higher costs than an alternative such as `c2.2xlarge`, which also provides similar runtime. Strategy c), while reasonable, might still lead to sub-optimal choices like `m3.xlarge`, which offers worse performance than `c3.2xlarge` for higher cost. Choosing a VM from over a 100 types across multiple cloud providers is even harder.

How does PARIS help? PARIS generates a performance-cost trade-off map with predictions of mean and p90 values of performance according to the user-specified performance metric and tailored to a user-specified task that represents her workload. Figure 5.14 shows such a trade-off map with predicted latencies (top) and corresponding *task completion costs* for a representative task consisting of a set of 225K YCSB queries on a Redis data-store. The p90 values are shown as error bars. The X-axis has different AWS and Azure VM types in an increasing order of their cost-per-hour. The reference VMs were `A2` and `F8` for Azure and `c4.2xlarge` and `m3.large` for AWS.

The user can use this map to choose the best VM for any performance and cost goals, then

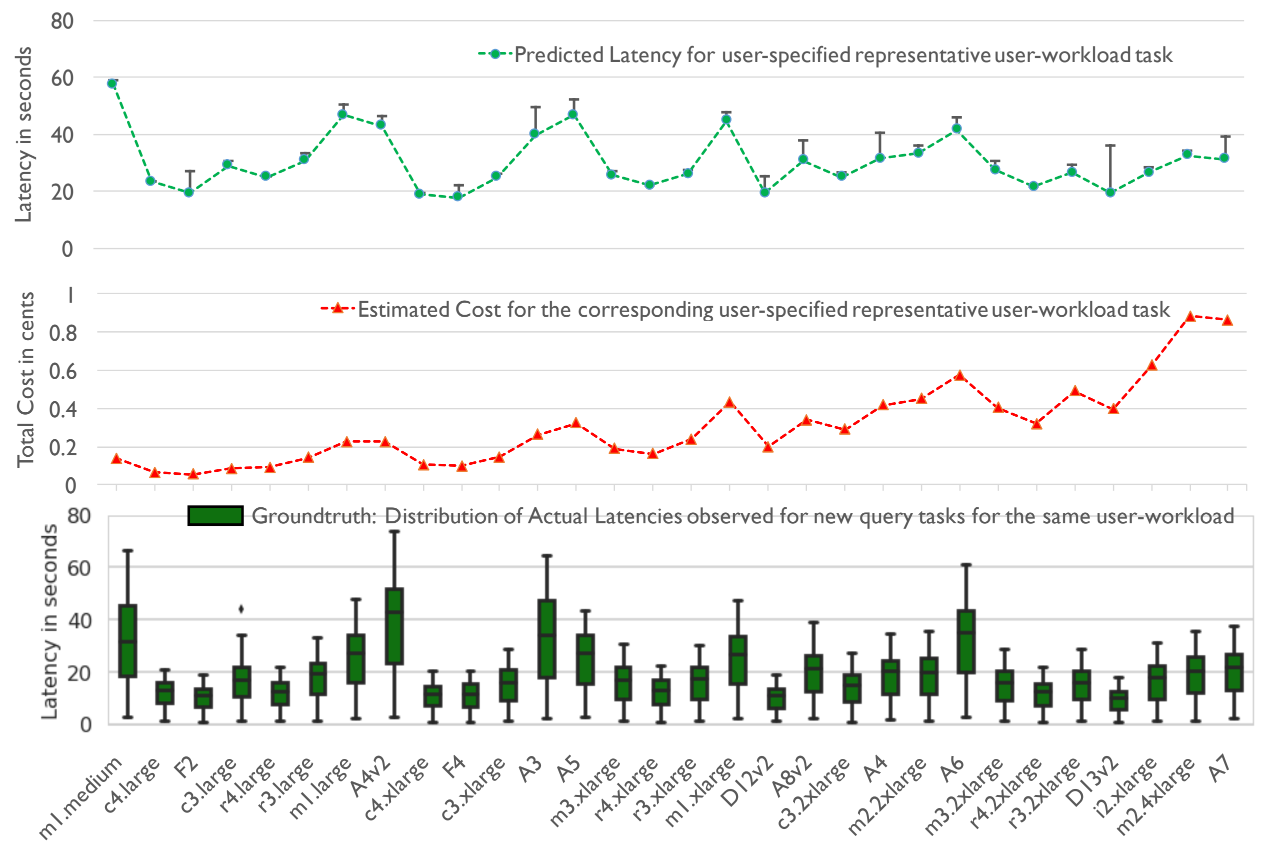


Figure 5.14: Performance-cost trade-off Map generated by PARIS using user-specified representative task that consisted of a set of 225K YCSB queries with a mix of 10/90 Reads/Writes, on a Redis data-store with 225K records. X-axis: AWS and Azure VM types ordered by increasing cost per hour. Reference VMs: `A2`, `F8` for Azure and `c4.2xlarge`, `m3.large` for AWS. **Top:** Predicted mean and p90 latencies (shown by whiskers). **Middle:** Estimated cost in cents for the representative task. **Bottom:** Distribution of actual observed latencies across different AWS and Azure VM types, for a set of 2.5K query user-tasks on Redis. (Sec. 5.5.5).

run their entire workload on the chosen VM. The last plot in Figure 5.14 shows the true latencies observed when *all* query tasks from the user workload are run on each VM. PARIS' predictions match these true latencies well. As before, the latencies do not directly correlate with the published cost-per-hour of the VMs; `F2`, for instance, achieves lower latencies than `A4v2`. PARIS predicts these counterintuitive facts correctly.

5.5.6 Quantifying cost savings

PARIS offers users considerable flexibility in choosing their own performance and cost goals. The precise gains a user gets from PARIS will depend on these goals. Nevertheless, below we consider two example policies that the user might follow, and quantify cost savings for each.

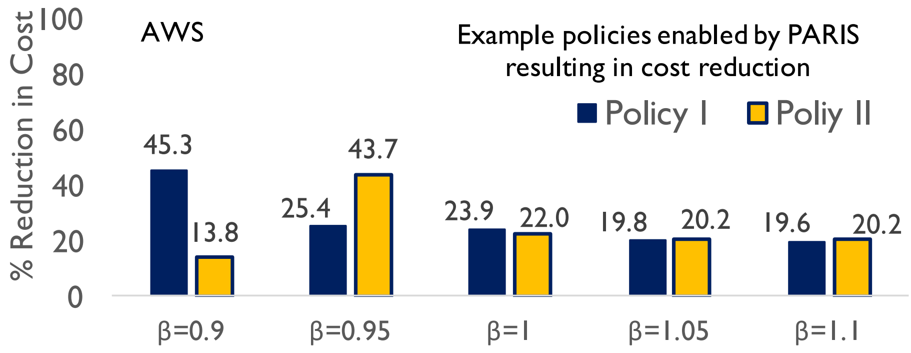


Figure 5.15: Percentage reduction in user costs enabled by PARIS' predictors over the baseline predictors on AWS for a number of policies. Policy I chooses the VM type with least predicted cost provided mean runtime $\leq \beta$ times the mean across VMs. Policy II is similar but thresholds $p90$ values instead. (Sec. 5.5.6.1).

5.5.6.1 Reduced user costs through better decisions

We performed this experiment on AWS, using YCSB-based serving workloads on Aerospike, MongoDB, Redis, and Cassandra data stores. We generated two performance-cost trade-off maps: one using predictions from PARIS and the other using Baseline$_2$ predictors. For each map, we chose a VM type for this workload using the policies described below, executed the workload on this VM type, and compared costs. We considered two example policies:

Policy I: Policy I picks the VM type with the least estimated cost provided the predicted runtime is less than a user-specified threshold, which is expressed as a fraction β of the mean predicted runtime across VM types.

Policy II: Instead of predicted runtime, Policy II uses predicted $p90$ to choose a VM type based on the same criterion. This policy optimizes for worst case performance.

We varied β in $[0.9, 1.1]$. As shown in Figure 5.15, the user can reduce costs by up to 45% by using performance and cost estimates from PARIS instead of the baseline.

5.5.6.2 Cost overheads of PARIS

PARIS does incur some limited overhead to produce the performance estimates. Part of this overhead is the *one-time* cost of offline benchmarking of VM types (see Table 5.1), which is amortized across all user workloads. The rest of the overhead is the cost of running a user-specified task on the reference VMs. As shown in Section 5.5.4.2, two reference VMs are enough for accurate predictions.

To quantify the cost overheads of PARIS empirically, we computed the cost of the offline VM benchmarking phase and the cost for fingerprinting each user-specified representative task in the online performance prediction phase. We compared this cost to the cost incurred by an alternative

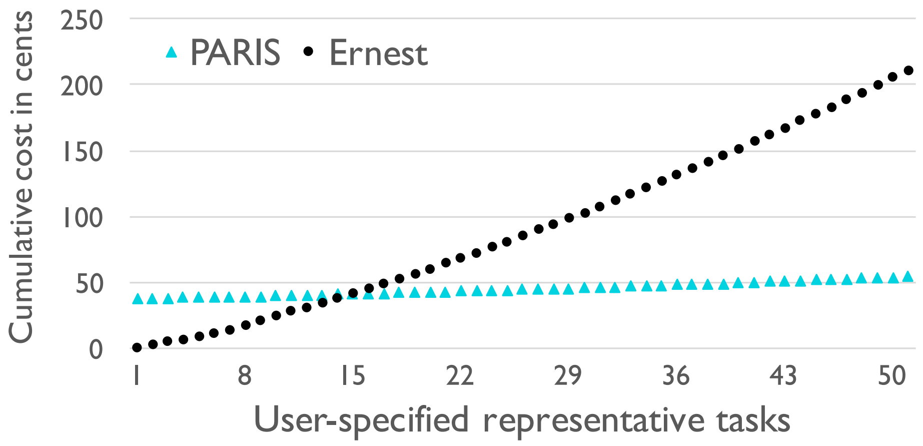


Figure 5.16: Cost overheads of PARIS compared to brute-force profiling on all VM types (e.g., in Ernest [152]).

that exhaustively runs the task on each VM type to choose the right VM type. This alternative strategy is what would be followed by systems like Ernest [152] (Ernest also performs additional profiling to determine the number of VMs; this is not included in our comparison). Figure 5.16 shows this comparison for the mean and p90 latency prediction task using core YCSB queries A and B as train and a set of 50 newly implemented workloads as the user-specified representative tasks. For this experiment, we used the cost of the VMs per unit time published by the cloud providers. We note that PARIS has a non-zero initial cost due to the offline VM benchmarking phase, but once this phase is over, the additional cost of fingerprinting each new user-specified task is much lower than the cost of Ernest's exhaustive search. Ernest's cumulative cost grows at a much higher rate than PARIS' and overtakes the latter after about 15 tasks. PARIS is therefore lightweight.

5.6 Limitations and Next Steps

PARIS took an initial step in helping users choose a right cloud VM type for their workloads. Here, we describe limitations of the current system and possible future directions. PARIS assumes the availability of a representative task from a user workload. Some tasks might be heavily input dependent; extending PARIS' modeling formulation to include task-specific features, such as input size, can enable generalization across tasks. While our current version requires separate fingerprints for each cloud provider, our modeling framework can be extended to use common fingerprints across multiple providers. PARIS is not aimed at estimating scaling behavior, but can be combined with approaches such as Ernest [152] that tackle the scaling problem. PARIS can also be extended to work with customizable VM sizes in the cloud, for instance, custom images by Google Cloud Engine [76].

5.7 Conclusion

In this chapter we presented PARIS, a system that allows users to choose the right VM type for meeting their performance goals and cost constraints through accurate and economical performance estimation. PARIS decouples the characterization of VM types from the characterization of workloads, thus eliminating the $O(n^2)$ cost of performance estimation while delivering accurate performance predictions across VM types from multiple cloud providers. We showed empirically that PARIS accurately predicts mean and tail performance for many realistic workloads and performance metrics across multiple clouds, and results in more cost effective decisions while meeting performance goals.

Chapter 6

Related Work

Resource management for computer systems is a deeply studied field of research. We now describe relevant prior work in straggler mitigation, cluster scheduling, resource allocation, and systems that use models with various objectives.

6.1 Cluster Schedulers

Assignment of tasks to nodes in a cluster has been studied widely [84, 53]. While most of these approaches mainly focus on improving the mean task slowdown, we aimed at improving the tail latencies for tasks. Recently proposed scheduling mechanisms optimize fairness [71, 166], and capacity [80], and data locality [168]. Delay-scheduling [168], which is a part of Hadoop's Fair-scheduler, suggests relaxing the fair queuing policy slightly to speed up the tasks by achieving data locality. Quincy [91] computes an optimal task-level schedule that balances fairness, data locality and freedom from starvation. Mesos [86], YARN [151], and Omega [134] propose schedulers that allocate resources at a coarse granularity that enables scheduling of jobs from different frameworks on a cluster. Sparrow [119], and Apollo [33] are distributed schedulers accommodating short-running tasks. Our work in predictive scheduling can be used along with these sophisticated schedulers to render predictable and faster jobs using fewer resources. Being learning-based, *Wrangler* uses more information about the nodes' resource usage than a conventional scheduler. It weighs them automatically to find and avoid situations that could cause tasks to run slower on underlying nodes.

Tarazu [3] proposes a communication-aware load-balancer for heterogeneous clusters consisting of high and low performance servers. Though the authors have mentioned that their work does not focus on stragglers, it may seem that as a side-effect this could benefit straggler avoidance. However, we have observed that nodes get temporarily overloaded even when all the nodes in a cluster are of the same type. Our predictive schedulers avoid creating stragglers by avoiding task-assignments to such temporarily overloaded nodes. Being focused on load balancing in heterogeneous clusters, Tarazu completely ignores this temporary overloading that occurs in the subset of similar types of servers. Wrangler builds models for individual nodes. These models consider more complex task-to-task and task-to-node interactions to avoid stragglers caused by temporary overloading of nodes.

6.2 Tackling Stragglers

We now describe recent work that directly focuses on straggler mitigation in distributed analytical systems.

6.2.1 Task Replication for Straggler *Mitigation*

Replicating work on idle resources has been shown to improve efficiency for certain workloads in distributed systems [69, 47, 23, 10]. Speculative execution [56] reacts to a straggler task by launching multiple copies of the task. Such reactive and replicative nature renders it wasteful in time and resources used to mitigate stragglers. Dolly [11] is a proactive and replication-based mechanism that improves completion time of interactive jobs. It ends up using extra resources for the replicated copies of tasks.

6.2.2 Early Detection of Stragglers

To reduce waiting in the observe phase, many approaches attempt to detect the slow running tasks as soon as possible and relaunch them [12, 90, 170, 14]. Most of the proposed techniques take a reactive approach since they assume that stragglers cannot be predicted. Multiple studies [15, 11] conclude that estimating task execution times in advance is challenging. Moreover, errors of models that predict task execution times are mostly intolerable as they could not only lengthen job completions but also affect resource utilization. In our work on predictive scheduling [160, 92, 161], we showed that node-level resource usage statistics can be used to predict if a task will run slow on a node even before launching it on that node. We showed that it is feasible to reduce the effect of straggler tasks in a proactive manner by using a predict-and-schedule mechanism.

6.2.3 Learning-based Approaches for Straggler *avoidance*

Recent literature has proposed learning for detecting stragglers [32] and modifying the schedulers to mitigate them [79]. Bortnikov, et al., [32], build a decision tree based on task and node level features to estimate slowdown of a task compared to similar tasks. These are the tasks with same input/output sizes or similar profiles. In our work, we learn to predict if a node is too busy to finish a task in a timely manner. We then close the loop, by modifying the scheduler to use these predictions and avoid creating stragglers. In our attempts in closing the loop, we realized that prediction errors can render the scheduling unreliable, resulting in long running jobs. To create a reliable scheduling mechanism, we attached confidence measures with our predictions. However, Bortnikov, et al., [32] did not demonstrate a scheduler that can use their predictor while being resilient to modeling errors.

ThroughputScheduler [79], is a scheduling mechanism for improving overall job completion times on a clusters of heterogeneous nodes that assigns jobs to nodes by optimally matching job requirements to node capabilities. ThroughputScheduler does not focus on mitigating stragglers. The authors evaluated their approach using Hadoop example workloads executed for a few minutes on a 5 node cluster. In our study, we observed that confidence measure is critical to guard the system against modeling errors. ThroughoutScheduler misses this difficulty at scale as it is evaluated on a small setup with example workloads.

6.3 Resource Allocation

There is a lot of prior work in resource allocation in batch and high-performance computing systems, operating systems, real-time computing, and more recently cluster and datacenter management. However, most of the work has focused on how to schedule tasks or jobs once their resource requirements are known. For instance, classical batch systems [65, 66, 5], and modern cluster management systems such as Eucalyptus [116], Condor [128], Hadoop [167, 81, 169], Quincy [91], and Mesos [86, 71], require users to specify their resource requirements. Users most often do not know the resource requirements, and find it hard to judge the performance and cost implication of their choices. Our work on PARIS allows users to specify performance and cost goals and chooses a VM type for user applications.

6.3.1 Interference Prediction

To estimate the implications of resource allocation on performance of applications, one class of related work focuses on quantifying or predicting interference between co-scheduled applications. Some approaches co-schedule applications with disjoint resource requirements [143, 135, 172, 175, 108, 109], while others use a trial-and-error approach [144, 106, 173]. Some approaches use measurements or performance models to predict interference among applications [77, 154, 153, 96, 150, 140, 57, 58].

Some approaches used dynamically monitored hardware-level features, such as CPI (Cycles Per Instruction) for interference prediction; however they aim at consolidating VMs on underlying physical machines [42, 104, 103]. Although these approaches use dynamic resource utilization statistics similar to PARIS, they mainly use only CPI and CMR (Cache Miss Rate), as compared to the fingerprint PARIS collects, consisting of about 40 different resource usage values collected from a couple of reference VMs. Moreover, CPI and CMR being hardware performance counters, are not as easily available as the VM-level counters that PARIS uses.

6.3.2 Performance prediction based on system modeling

A lot of prior work predicts performance based on the properties of the system and patterns of workloads. A variety of areas, such as databases [39, 112], scientific computing [48], and programming models [28] have used this approach.

To choose the right VM size, Pseudoapp [142] creates a pseudo-application with the same set of distributed components and executes the same sequence of system calls as those of the real application. However, PseudoApp needs a complete knowledge of what a real application is doing, which may not always be available. Ernest [152] builds a model to predict the runtime of distributed analytics jobs as a function of cluster size. While Ernest adopts a similar model based approach, their modeling framework is incapable of inferring the performance of new workloads on a VM type without first running the workload on that VM type. Cherrypick [4] uses Bayesian optimization to figure out the best resource configuration for applications. However, Cherrypick requires to run an application on 10 to 20 configurations guided by the acquisition function before estimating performance across the whole spectrum of cloud configurations.

6.3.3 Adaptive control systems

Another approach to predicting performance is adaptive control using feedback-driven systems such as controllers or reinforcement learning. Sometimes this is used in combination with offline or online performance models. The goal is usually to dynamically allocate resources. For example, Rightscale [89] for EC2 adapts to the changing load of applications by adding VM instances. Other systems have explicit models to better inform the control system, e.g., [29, 67, 114]. Feedback is also used in Quasar [58] to update estimates of the sensitivity of application's performance to heterogeneity, interference, scale-up and scale-out of resources in the given cluster. This dissertation demonstrates how to close the loop when using performance prediction models to derive resource allocation decisions. By closing the loop, we allow the models to adapt to the dynamic, highly variable execution environments.

6.3.4 Workload Forecasting

To enable proactive resource allocation, some prior work proposes mechanisms to forecast workload changes. PRESS [73] dynamically adapts resources allocated to an application allowing cloud-providers to meet the usual bi-objective: minimize resource provisioning costs, and satisfy SLOs. Even though the paper focused on CPU usage, the authors claim that the proposed mechanism can be extended to other resource types. Applications may have different resource bottlenecks. Adding or removing memory may have different implications on performance of an application that adding or removing CPU cores. Our work allows users to obtain performance-cost trade-off maps tailored to their workloads, allowing them to choose the resources that suit their needs. Gmach, et al., [72] use traces of workloads to generate synthetic workloads and predict future resource needs. They use their system to aid capacity planning in datacenters. Soror, et al., [138] use information about the expected workload of a database to create workload-specific VM configurations. Their framework requires the database management system to represent the workload as a set of SQL statements. Our work alleviates the need for significant knowledge about the application. Using fingerprints collected by running a task from a workload on a small number of reference VMs, we predict the performance-cost trade-off tailored to that workload across various options in the public cloud environments.

Chapter 7

Conclusions and Future Directions

In this dissertation, we argued for automated resource management for datacenters and cloud-hosted systems using Machine Learning techniques.

With **Wrangler (Chapter 3)**, we designed, implemented and evaluated a predictive scheduler for improved and predictable job completion times while reducing total resources used. We formulated the problem of straggler avoidance as an instance of binary classification. Wrangler introduced a notion of confidence measure attached to predictions that served as a key to guard the system against wrong predictions or modeling errors.

Through our **Multi-task Learning formulations (Chapter 4)**, we showed how to reduce the cost of training-data capture time by sharing relevant data across similar modeling tasks. We also showed how we can gain more insights into the models' learning by extending our formulations using group sparsity inducing norms.

Through **PARIS (Chapter 5)**, we demonstrated how to learn from benchmark workloads and estimate performance of real user workloads in the public cloud environments. PARIS' data-driven approach used a novel hybrid offline and online data collection and modeling framework enabling accurate performance prediction with minimal data collection.

There is a lot left to be done to achieve the vision of automatic resource management using statistical learning based models. Also, a lot of new avenues of research will open up as we achieve this vision. Before concluding, we discuss upcoming and potential future directions.

7.1 Avenues for future directions in systems research

Serverless computing with performance guarantees With the evolution in computing, resource infrastructure has evolved from physical servers, to virtual servers in datacenters, to virtual servers in the cloud, to serverless computing frameworks. We see more and more automation in provisioning of resources as we follow this evolution. Serverless computing has many attractive features, including no need to manage servers, continuous scaling, and sub-second metering. However, with increased automation, users have simultaneously gotten reduced control over the underlying physical resources, rendering performance and in turn resulting cost hard to reason about.

We envision a serverless computing framework with an interface that allows users all the benefits of automated resource provisioning while allowing them to specify their performance and cost requirements or Service Level Objectives (SLOs). Exploring this research opportunity needs to deal with various challenges such as deciding on the desirable granularity of computations to allow predictability of performance, understanding performance as a function of allocated resources, and characteristics of functions executed on different inputs.

Automatic Resource Provisioning in the cloud-middle-edge spectrum With the enormous increase in data, and the evolution of cloud computing, code is most likely not co-located with the source of data generation. This shift from local or on-prem datacenters to cloud-hosted distributed systems was facilitated by many benefits such as scalability, reduced costs, and automatic fault tolerance the core, or the cloud, has to offer. However, the data transfer latency in core computing poses challenges to emerging new applications, such as self-driven cars, smart building, and other applications in the field of IoT, where there is a need to make decisions by processing large amounts of data in real time. Edge computing, where compute is close to the source of data generation, allows latency sensitive applications to make decisions in realtime.

The core [17, 56] is rich in resources and services, but it involves data transfer latency. Whereas by computing near the source of data, edge [45, 132] removes the hurdle of transfer latency, but has limited resources. There are approaches that also allow compute and storage resources to be somewhere in the middle [88, 30], away from the core and the edge. Each of these, the core, the middle, and the edge, have advantages and disadvantages. Putting these together in such a way that allows us to strengthen their advantages while reducing the disadvantages will benefit both existing and emerging new applications. How to adaptively allocate resources across the core-middle-edge spectrum to enable a transparent interface to the users, while providing performance guarantees is a vital open research question.

7.2 Avenues for future directions in ML research

Dealing with change In addition to the field of resource management discussed in this dissertation, Machine Learning techniques are being deployed in a wide range of applications in which new data is continuously collected. In each application there are essential questions around when and how to retrain models as the world changes and how to ensure that the appropriate data is collected. While advances in change-point detection, online learning, and bandit techniques, address various aspects of this problem, integrating all of these techniques with existing models and training algorithms can be challenging, and needs further work.

Interpretability of models When applying ML models for deriving important decisions in the domain of distributed systems, in addition to accurate predictions, it is helpful for domain experts to gain insights into the decision making process of the models. The need for explainability has forced many existing model-driven systems to choose simple models such as decision trees. More powerful and popular models today, especially deep learning based models, are hard to understand. To further our vision of model-based automatic resource management, we will need to think of building models that achieve both, high accuracy and interpretability.

7.3 Concluding Remarks

In this dissertation, we demonstrated the impact of using ML for automatic resource management for distributed systems, by carefully dealing with the challenges they raise, such as model uncertainty, cost of training, generalization from benchmark workloads to real user workloads, and interpretability of the models. We discussed challenges that are left to be answered including dealing with change in data distributions, and building more powerful interpretable models.

Finally, it is important to understand when we should use ML models in the context of systems. Based on our experience, learning can be employed when either the scale or the complex dynamic nature of the problem dominates the decision-making process. Heuristics abound in computer systems. They work well for the common cases. However, with the evolution in computing and the growing complexity of resource infrastructure, we are faced with situations that demand custom decisions. Heuristics need to be retuned for such environments. ML models, unlike heuristics, can derive actions from rich runtime context information. Extensive use of learning enables systems that are agile to dynamic changes in workloads and execution environments. We demonstrated how we leveraged learning techniques for making scheduling and resource allocation decisions for the cloud and the datacenters. *Going forward, we envision ML models enabling better decisions anywhere we are using heuristics in computer systems.*

Bibliography

[1] *Aerospike Datastore*. https://www.aerospike.com. 2017.

[2] Jonathan Aflalo et al. "Variable Sparsity Kernel Learning." In: *Journal of Machine Learning Research* 12 (2011).

[3] Faraz Ahmad et al. "Tarazu: Optimizing MapReduce on Heterogeneous Clusters". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 61–74. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150984. URL: http://doi.acm.org/10.1145/2150976.2150984.

[4] Omid Alipourfard et al. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard.

[5] Gail Alverson et al. "Scheduling on the Tera MTA". In: *In Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1995, pp. 19–44.

[6] *Amazon EC2*. https://aws.amazon.com/ec2. 2017.

[7] *Amazon Redshift*. http://aws.amazon.com/redshift/faqs/.

[8] *Amazon Web Services*. https://aws.amazon.com/. 2017.

[9] Amazon.com. *Amazon Web Services: Case Studies*. https://aws.amazon.com/solutions/case-studies/. 2017.

[10] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. "Conflict-aware Scheduling for Dynamic Content Applications". In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS'03. Seattle, WA: USENIX Association, 2003, pp. 6–6. URL: http://dl.acm.org/citation.cfm?id=1251460.1251466.

[11] Ganesh Ananthanarayanan et al. "Effective Straggler Mitigation: Attack of the Clones". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. NSDI'13. Lombard, IL: USENIX Association, 2013, pp. 185–198. URL: http://dl.acm.org/citation.cfm?id=2482626.2482645.

[12] Ganesh Ananthanarayanan et al. "GRASS: Trimming Stragglers in Approximation Analytics". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 289–302. ISBN: 978-1-931971-09-6. URL: `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/ananthanarayanan`.

[13] Ganesh Ananthanarayanan et al. "PACMan: Coordinated Memory Caching for Parallel Jobs". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 20–20. URL: `http://dl.acm.org/citation.cfm?id=2228298.2228326`.

[14] Ganesh Ananthanarayanan et al. "Reining in the Outliers in Map-reduce Clusters Using Mantri". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–16. URL: `http://dl.acm.org/citation.cfm?id=1924943.1924962`.

[15] David P. Anderson et al. "SETI@home: an experiment in public-resource computing". In: *Commun. ACM* 45.11 (Nov. 2002), pp. 56–61. ISSN: 0001-0782. DOI: `10.1145/581571.581573`. URL: `http://doi.acm.org/10.1145/581571.581573`.

[16] Rie Kubota Ando and Tong Zhang. "A framework for learning predictive structures from multiple tasks and unlabeled data". In: *JMLR* 6 (2005).

[17] Michael Armbrust et al. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. 2009.

[18] *AWS Customer Success*. `https://aws.amazon.com/solutions/case-studies/`. 2017.

[19] *AWS Lambda*. `https://aws.amazon.com/lambda/`. 2017.

[20] *Azure Functions*. `https://azure.microsoft.com/en-us/services/functions/`. 2017.

[21] Francis Bach et al. "Convex optimization with sparsity-inducing norms". In: *Optimization for Machine Learning* (2011), pp. 19–53.

[22] Francis R Bach. "Consistency of the group lasso and multiple kernel learning". In: *The Journal of Machine Learning Research* 9 (2008), pp. 1179–1225.

[23] Henri E. Bal et al. "Replication techniques for speeding up parallel applications on distributed systems." In: *Concurrency - Practice and Experience* 4.5 (Nov. 30, 2007), pp. 337–355. URL: `http://dblp.uni-trier.de/db/journals/concurrency/concurrency4.html#BalKTJ92`.

[24] Jonathan Baxter. "A model of inductive bias learning". In: *JAIR* 12 (2000).

[25] Sarah Bird. "Optimizing Resource Allocations for Dynamic Interactive Applications". PhD thesis. EECS Department, University of California, Berkeley, 2014.

[26] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 0387310738.

[27] Gilles Blanchard, Gyemin Lee, and Clayton Scott. "Generalizing from several related classification tasks to a new unlabeled sample". In: *NIPS*. 2011.

[28] Guy E. Blelloch. "Programming Parallel Algorithms". In: *Commun. ACM* 39.3 (1996), pp. 85–97. ISSN: 0001-0782.

[29] Peter Bodik et al. "Automatic Exploration of Datacenter Performance Regimes". In: *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*. ACDC '09. Barcelona, Spain: ACM, 2009, pp. 1–6. ISBN: 978-1-60558-585-7. DOI: 10.1145/1555271.1555273. URL: http://doi.acm.org/10.1145/1555271.1555273.

[30] Flavio Bonomi et al. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: ACM, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. URL: http://doi.acm.org/10.1145/2342509.2342513.

[31] Antoine Bordes et al. "Fast Kernel Classifiers with Online and Active Learning". In: *J. Mach. Learn. Res.* 6 (Dec. 2005), pp. 1579–1619. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1046920.1194898.

[32] Edward Bortnikov et al. "Predicting Execution Bottlenecks in Map-reduce Clusters". In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'12. Boston, MA: USENIX Association, 2012, pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=2342763.2342781.

[33] Eric Boutin et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 285–300. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685071.

[34] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 0521833787.

[35] Leo Breiman. "Random Forests". In: *Mach. Learn.* (Oct. 2001). ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: http://dx.doi.org/10.1023/A:1010933404324.

[36] Christopher J. C. Burges. "A Tutorial on Support Vector Machines for Pattern Recognition". In: *Data Min. Knowl. Discov.* 2.2 (June 1998), pp. 121–167. ISSN: 1384-5810. DOI: 10.1023/A:1009715923555. URL: http://dx.doi.org/10.1023/A:1009715923555.

[37] Josiah L. Carlson. *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013. ISBN: 1617290858, 9781617290855.

[38] Rich Caruana. “Multitask Learning: A Knowledge-Based Source of Inductive Bias”. In: *ICML*. 1993.

[39] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. “Estimating Progress of Execution for SQL Queries”. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’04. 2004. ISBN: 1-58113-859-8.

[40] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. “Editorial: Special Issue on Learning from Imbalanced Data Sets”. In: *SIGKDD Explor. Newsl.* 6.1 (June 2004), pp. 1–6. ISSN: 1931-0145. DOI: 10.1145/1007730.1007733. URL: http://doi.acm.org/10.1145/1007730.1007733.

[41] Nitesh V. Chawla et al. “SMOTE: Synthetic Minority Over-sampling Technique”. In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357.

[42] Xi Chen et al. “CloudScope: Diagnosing Performance Interference for Resource Management in Multi-Tenant Clouds”. In: *23rd IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*. Atlanta, GA, USA, 2015.

[43] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. “Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads.” In: *PVLDB* 5.12 (2012), pp. 1802–1813. URL: http://dblp.uni-trier.de/db/journals/pvldb/pvldb5.html#ChenAK12.

[44] Yanpei Chen et al. “The Case for Evaluating MapReduce Performance Using Workload Suites”. In: *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS ’11. Washington, DC, USA: IEEE Computer Society, https://github.com/SWIMProjectUCB/SWIM/wiki/., 2011. ISBN: 978-0-7695-4430-4. DOI: 10.1109/MASCOTS.2011.12. URL: SWIMcanbedownloadedfromhttps://github.com/SWIMProjectUCB/SWIM/wiki/..

[45] Zhuo Chen et al. “An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance”. In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. SEC ’17. San Jose, California: ACM, 2017, 14:1–14:14. ISBN: 978-1-4503-5087-7. DOI: 10.1145/3132211.3134458. URL: http://doi.acm.org/10.1145/3132211.3134458.

[46] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. 1st. O’Reilly Media, Inc., 2010. ISBN: 1449381561, 9781449381561.

[47] Walfredo Cirne et al. “On the efficacy, efficiency and emergent behavior of task replication in large distributed systems”. In: *Parallel Comput.* 33.3 (Apr. 2007). ISSN: 0167-8191. DOI: 10.1016/j.parco.2007.01.002. URL: http://dx.doi.org/10.1016/j.parco.2007.01.002.

[48] Rice University. Department of Computer Science et al. *A Static Performance Estimator to Guide Data Partitioning Decisions*. 136. Rice University, Department of Computer Science, 1990.

[49] Michael Conley, Amin Vahdat, and George Porter. "Achieving Cost-efficient, Data-intensive Computing in the Cloud". In: SoCC '15. Kohala Coast, Hawaii, 2015. ISBN: 978-1-4503-3651-2.

[50] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152. URL: http://doi.acm.org/10.1145/1807128.1807152.

[51] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995).

[52] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. New York, NY, USA: Cambridge University Press, 2000. ISBN: 0-521-78019-5.

[53] Mark Crovella, Mor Harchol-Balter, and Cristina D. Murta. "Task Assignment in a Distributed System: Improving Performance by Unbalancing Load (Extended Abstract)." In: *SIGMETRICS*. Dec. 6, 2002, pp. 268–269.

[54] Barroso L. Dean Jeff. *Achieving Rapid Response Times in Large Online Services*. http://research.google.com/people/jeff/latency.html, 2012.

[55] Jeffrey Dean and Luiz André Barroso. "The Tail at Scale". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 74–80. ISSN: 0001-0782. DOI: 10.1145/2408776.2408794. URL: http://doi.acm.org/10.1145/2408776.2408794.

[56] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[57] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters". In: *ASPLOS*. Houston, TX, USA, 2013.

[58] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541941. URL: http://doi.acm.org/10.1145/2541940.2541941.

[59] David L Donoho. "For most large underdetermined systems of linear equations the minimal l1-norm solution is also the sparsest solution". In: *Communications on pure and applied mathematics* 59.6 (2006).

[60] Hikmet Dursun et al. "An MPI Performance Monitoring Interface for Cell Based Compute Nodes". In: *Parallel Processing Letters* 19.4 (2009).

[61] Theodoros Evgeniou and Massimiliano Pontil. "Regularized multi–task learning". In: *KDD*. 2004.

[62] Theodoros Evgeniou et al. "Learning multiple tasks with kernel methods." In: *JMLR* 6.4 (2005).

[63] Rong-En Fan et al. "LIBLINEAR: A library for large linear classification". In: *Journal of Machine Learning Research* 9 (2008).

[64] Benjamin Farley et al. "More for Your Money: Exploiting Performance Heterogeneity in Public Clouds". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 20:1–20:14. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391249. URL: http://doi.acm.org/10.1145/2391229.2391249.

[65] Dror G. Feitelson. *Job Scheduling in Multiprogrammed Parallel Systems*. 1997.

[66] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. "Parallel Job Scheduling - A Status Report". In: *JSSPP*. 2004, pp. 1–16.

[67] Andrew D. Ferguson et al. "Jockey: Guaranteed Job Latency in Data Parallel Clusters". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 99–112. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168847. URL: http://doi.acm.org/10.1145/2168836.2168847.

[68] *Ganglia*. https://sourceforge.net/projects/ganglia/. 2016.

[69] Gaurav D. Ghare and Scott T. Leutenegger. "Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW". In: *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing*. JSSPP'04. New York, NY: Springer-Verlag, 2005, pp. 264–287. ISBN: 3-540-25330-0, 978-3-540-25330-3. DOI: 10.1007/11407522_15. URL: http://dx.doi.org/10.1007/11407522_15.

[70] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *SOSP*. 2003.

[71] Ali Ghodsi et al. "Dominant resource fairness: fair allocation of multiple resource types". In: NSDI'11. Boston, MA, 2011.

[72] Daniel Gmach et al. "Workload Analysis and Demand Prediction of Enterprise Data Center Applications". In: IISWC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 171–180. ISBN: 978-1-4244-1561-8. DOI: 10.1109/IISWC.2007.4362193. URL: http://dx.doi.org/10.1109/IISWC.2007.4362193.

[73] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. “PRESS: PRedictive Elastic ReSource Scaling for cloud systems.” In: *CNSM*. IEEE, 2010, pp. 9–16. ISBN: 978-1-4244-8908-4. URL: http://dblp.uni-trier.de/db/conf/cnsm/cnsm2010.html#GongGW10.

[74] Joseph E. Gonzalez et al. “GraphX: Graph Processing in a Distributed Dataflow Framework”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI’14. Broomfield, CO: USENIX Association, 2014, pp. 599–613. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685096.

[75] *Google Cloud Functions*. https://cloud.google.com/functions/. 2017.

[76] *Google Cloud Platform*. https://cloud.google.com/compute/. 2017.

[77] Sriram Govindan et al. “Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines”. In: SOCC ’11. Cascais, Portugal, 2011, 22:1–22:14. ISBN: 978-1-4503-0976-9.

[78] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. 1st. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011. ISBN: 0132091518, 9780132091510.

[79] Shekhar Gupta et al. “ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters”. In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC’13)*. San Jose, CA: USENIX, 2013, pp. 159–165. ISBN: 978-1-931971-02-7. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/gupta.

[80] *Hadoop’s Capacity Scheduler*. http://hadoop.apache.org/core/docs/current/capacity_scheduler.html. 2012.

[81] *Hadoop’s Capacity Scheduler*. http://hadoop.apache.org/core/docs/current/capacity_scheduler.html. 2017.

[82] Mohammad Hajjat et al. “Application-specific configuration selection in the cloud: impact of provider policy and potential of systematic testing”. In: *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE. 2015, pp. 873–881.

[83] Mark Hall et al. “The WEKA data mining software: an update”. In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009). ISSN: 1931-0145. DOI: 10.1145/1656274.1656278. URL: http://doi.acm.org/10.1145/1656274.1656278.

[84] Mor Harchol-Balter. “Task assignment with unknown duration”. In: *J. ACM* 49.2 (2002), pp. 260–288.

[85] Keqiang He et al. “Next Stop, the Cloud: Understanding Modern Web Service Deployment in EC2 and Azure”. In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC ’13. Barcelona, Spain: ACM, 2013, pp. 177–190. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504740. URL: http://doi.acm.org/10.1145/2504730.2504740.

[86] Benjamin Hindman et al. “Mesos: A Platform for Fine-grained Resource Sharing in the Data Center”. In: NSDI’11. Boston, MA: USENIX Association, 2011. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[87] Paul G. Hoel, Sidney C. Port, and Charles J. Stone. *Introduction to Stochastic Processes*. Boston, MA: Houghton Mifflin Company, 1972.

[88] Pengfei Hu et al. “Survey on Fog Computing”. In: *J. Netw. Comput. Appl.* 98.C (Nov. 2017), pp. 27–42. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2017.09.002. URL: https://doi.org/10.1016/j.jnca.2017.09.002.

[89] Rightscale Inc. *Amazon EC2: Rightscale*. http://www.rightscale.com/. 2017.

[90] Michael Isard et al. “Dryad: Distributed Data-parallel Programs from Sequential Building Blocks”. In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys ’07. Lisbon, Portugal: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273005. URL: http://doi.acm.org/10.1145/1272996.1273005.

[91] Michael Isard et al. “Quincy: Fair Scheduling for Distributed Computing Clusters”. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: ACM, 2009, pp. 261–276. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629601. URL: http://doi.acm.org/10.1145/1629575.1629601.

[92] Neeraja J. Yadwadkar et al. “Faster Jobs in Distributed Data Processing using Multi-Task Learning”. In: June 2015, pp. 532–540. ISBN: 978-1-61197-401-0.

[93] Laurent Jacob, Jean philippe Vert, and Francis R. Bach. “Clustered Multi-Task Learning: A Convex Formulation”. In: *NIPS*. 2009.

[94] Virajith Jalaparti et al. “Bridging the Tenant-provider Gap in Cloud Services”. In: SoCC ’12. San Jose, California, 2012. ISBN: 978-1-4503-1761-0.

[95] Marius Kloft et al. “lp-Norm Multiple Kernel Learning”. In: *J. Mach. Learn. Res.* 12 (July 2011), pp. 953–997. ISSN: 1532-4435.

[96] Younggyun Koh et al. “An analysis of performance interference effects in virtual environments”. In: *ISPASS*. 2007.

[97] S.P. T. Krishnan and Jose Ugia Gonzalez. *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*. 1st. Berkely, CA, USA: Apress, 2015. ISBN: 1484210050, 9781484210055.

[98] YongChul Kwon et al. “SkewTune: Mitigating Skew in Mapreduce Applications”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’12. Scottsdale, Arizona, USA: ACM, 2012, pp. 25–36. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213840. URL: http://doi.acm.org/10.1145/2213836.2213840.

[99] YongChul Kwon et al. “SkewTune: Mitigating Skew in Mapreduce Applications”. In: *SIGMOD*. 2012.

[100] Avinash Lakshman and Prashant Malik. “Cassandra: A Decentralized Structured Storage System”. In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010).

[101] Gunho Lee et al. “Topology-aware Resource Allocation for Data-intensive Workloads”. In: *SIGCOMM Comput. Commun. Rev.* 41.1 (Jan. 2011), pp. 120–124. ISSN: 0146-4833. DOI: 10.1145/1925861.1925881. URL: http://doi.acm.org/10.1145/1925861.1925881.

[102] Henry Li. *Introducing Windows Azure*. Berkely, CA, USA: Apress, 2009. ISBN: 143022469X, 9781430224693.

[103] Amiya K. Maji et al. “Mitigating Interference in Cloud Services by Middleware Reconfiguration”. In: *Proceedings of the 15th International Middleware Conference*. Middleware ’14. 2014.

[104] Amiya Kumar Maji, Subrata Mitra, and Saurabh Bagchi. “ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services.” In: *ICAC*. 2015.

[105] Grzegorz Malewicz et al. “Pregel: A System for Large-scale Graph Processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: http://doi.acm.org/10.1145/1807167.1807184.

[106] Jason Mars et al. “Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations”. In: MICRO-44. Porto Alegre, Brazil, 2011, pp. 248–259. ISBN: 978-1-4503-1053-6.

[107] Matthew L. Massie, Brent N. Chun, and David E. Culler. “The Ganglia Distributed Monitoring System: Design, Implementation And Experience”. In: *Parallel Computing* 30 (2003), p. 2004.

[108] Andreas Merkel and Frank Bellosa. “Balancing Power Consumption in Multiprocessor Systems”. In: EuroSys ’06. Leuven, Belgium, 2006, pp. 403–414. ISBN: 1-59593-322-0.

[109] Andreas Merkel and Frank Bellosa. “Task activity vectors: a new metric for temperature-aware scheduling”. In: *Proc. Eurosys ’08*. Glasgow, Scotland UK, 2008, pp. 1–12.

[110] Gordon E. Moore. “Readings in Computer Architecture”. In: ed. by Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Chap. Cramming More Components Onto Integrated Circuits, pp. 56–59. ISBN: 1-55860-539-8. URL: http://dl.acm.org/citation.cfm?id=333067.333074.

[111] Timothy Prickett Morgan. *Cluster Sizes Reveal Hadoop Maturity Curve*. URL: http://www.enterprisetech.com/2013/11/08/cluster-sizes-reveal-hadoop-maturity-curve/.

[112] Kristi Morton, Magdalena Balazinska, and Dan Grossman. “ParaTimer: A Progress Indicator for MapReduce DAGs”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. Indianapolis, Indiana, USA, 2010, pp. 507–518. ISBN: 978-1-4503-0032-2.

[113] Derek G. Murray et al. “Naiad: A Timely Dataflow System”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 439–455. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522738. URL: http://doi.acm.org/10.1145/2517349.2522738.

[114] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. “Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds”. In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys ’10. Paris, France, 2010, pp. 237–250. ISBN: 978-1-60558-577-2.

[115] Jan Newmarch. “FFmpeg/Libav”. In: *Linux Sound Programming*. Berkeley, CA: Apress, 2017, pp. 227–234. ISBN: 978-1-4842-2496-0. DOI: 10.1007/978-1-4842-2496-0_12. URL: http://dx.doi.org/10.1007/978-1-4842-2496-0_12.

[116] Daniel Nurmi et al. “The Eucalyptus Open-Source Cloud-Computing System”. In: CCGRID ’09. 2009, pp. 124–131. ISBN: 978-0-7695-3622-4.

[117] Zhonghong Ou et al. “Exploiting Hardware Heterogeneity Within the Same Instance Type of Amazon EC2”. In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*. HotCloud’12. Boston, MA: USENIX Association, 2012, pp. 4–4. URL: http://dl.acm.org/citation.cfm?id=2342763.2342767.

[118] Kay Ousterhout et al. “Performance clarity as a first-class design principle”. In: *16th Workshop on Hot Topics in Operating Systems (HotOS’17)*. 2017.

[119] Kay Ousterhout et al. “Sparrow: Distributed, Low Latency Scheduling”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 69–84. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522716. URL: http://doi.acm.org/10.1145/2517349.2522716.

[120] Shibin Parameswaran and Kilian Q. Weinberger. “Large Margin Multi-Task Metric Learning”. In: *NIPS*. 2010.

[121] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *J. Mach. Learn. Res.* 12 (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435.

[122] *Performance Monitor*. https://technet.microsoft.com/en-us/library/bb490957.aspx.

[123] John Platt. “Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods”. In: *Advances in large margin classifiers* 10.3 (1999).

[124] John C. Platt. “Fast Training of Support Vector Machines Using Sequential Minimal Optimization”. In: *Advances in Kernel Methods*. Ed. by Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola. Cambridge, MA, USA: MIT Press, 1999, pp. 185–208. ISBN: 0-262-19416-3. URL: http://dl.acm.org/citation.cfm?id=299094.299105.

[125] Ting Kei Pong et al. “Trace norm regularization: reformulations, algorithms, and multi-task learning”. In: *SIAM Journal on Optimization* 20.6 (2010).

[126] Ariadna Quattoni, Michael Collins, and Trevor Darrell. “Transfer learning for image classification with sparse prototype representations”. In: *CVPR*. 2008.

[127] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011. ISBN: 1107015359, 9781107015357.

[128] Rajesh Raman, Miron Livny, and Marv Solomon. “Matchmaking: An extensible framework for distributed resource management”. In: *Cluster Computing* 2.2 (Apr. 1999). ISSN: 1386-7857.

[129] Charles Reiss et al. “Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis”. In: SoCC ’12. San Jose, California. ISBN: 978-1-4503-1761-0.

[130] Bernardino Romera-Paredes et al. “Multilinear multitask learning”. In: *ICML*. 2013.

[131] Sheldon M. Ross. *Introduction to Probability Models, Eighth Edition*. 8th ed. Academic Press, Jan. 2003. ISBN: 0125980558. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0125980558.

[132] Mahadev Satyanarayanan. “The Emergence of Edge Computing”. In: *Computer* 50.1 (Jan. 2017), pp. 30–39. ISSN: 0018-9162. DOI: 10.1109/MC.2017.9. URL: https://doi.org/10.1109/MC.2017.9.

[133] Robert R. Schaller. “Moore’s Law: Past, Present, and Future”. In: *IEEE Spectr.* 34.6 (June 1997), pp. 52–59. ISSN: 0018-9235. DOI: 10.1109/6.591665. URL: http://dx.doi.org/10.1109/6.591665.

[134] Malte Schwarzkopf et al. “Omega: Flexible, Scalable Schedulers for Large Compute Clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. Prague, Czech Republic: ACM, 2013, pp. 351–364. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465386. URL: http://doi.acm.org/10.1145/2465351.2465386.

[135] Kai Shen et al. “Hardware counter driven on-the-fly request signatures”. In: *SIGOPS Oper. Syst. Rev.* 42.2 (2008), pp. 189–200. ISSN: 0163-5980.

[136] M Signoretto et al. “Graph Based Regularization for Multilinear Multitask Learning”. In: (2014).

[137] Tom Simonite. *MIT Technology Review, Moore?s Law Is Dead. Now What?* URL: `https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/`.

[138] Ahmed A. Soror et al. “Automatic Virtual Machine Configuration for Database Workloads”. In: SIGMOD ’08. Vancouver, Canada: ACM, 2008, pp. 953–966. ISBN: 978-1-60558-102-6. DOI: `10.1145/1376616.1376711`. URL: `http://doi.acm.org/10.1145/1376616.1376711`.

[139] *Squash Compression Benchmark.* `https://quixdb.github.io/squash-benchmark/`. 2017.

[140] Christopher Stewart, Terence Kelly, and Alex Zhang. “Exploiting Nonstationarity for Performance Prediction”. In: EuroSys ’07. Lisbon, Portugal, 2007, pp. 31–44. ISBN: 978-1-59593-636-3.

[141] Yanmin Sun et al. “Cost-sensitive Boosting for Classification of Imbalanced Data”. In: *Pattern Recogn.* 40.12 (Dec. 2007), pp. 3358–3378. ISSN: 0031-3203. DOI: `10.1016/j.patcog.2007.04.009`. URL: `http://dx.doi.org/10.1016/j.patcog.2007.04.009`.

[142] Byung-Chul Tak et al. “PseudoApp: Performance prediction for application migration to cloud”. In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27-31, 2013*. 2013, pp. 303–310.

[143] David Tam, Reza Azimi, and Michael Stumm. “Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors”. In: *EuroSys ’07*. Lisbon, Portugal, 2007. ISBN: 978-1-59593-636-3.

[144] Lingjia Tang et al. “The Impact of Memory Subsystem Resource Sharing on Datacenter Applications”. In: ISCA ’11. San Jose, California, USA, 2011, pp. 283–294. ISBN: 978-1-4503-0472-6.

[145] Sebastian Thrun. “Is learning the n-th thing any easier than learning the first?” In: *NIPS* (1996).

[146] Ashish Thusoo et al. “Hive: A Warehousing Solution over a Map-reduce Framework”. In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1626–1629. ISSN: 2150-8097.

[147] Robert Tibshirani. “Regression shrinkage and selection via the lasso”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), pp. 267–288.

[148] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Berlin, Heidelberg: Springer-Verlag, 1995. ISBN: 0-387-94559-8.

[149] M. Varma and D. Ray. “Learning The Discriminative Power-Invariance Trade-Off”. In: *ICCV*. 2007.

[150] Nedeljko Vasic et al. “DejaVu: accelerating resource allocation in virtualized environments”. In: *ASPLOS*. 2012, pp. 423–436.

[151] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: SOCC ’13. Santa Clara, California. ISBN: 978-1-4503-2428-1.

[152] Shivaram Venkataraman et al. “Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics”. In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 363–378. ISBN: 978-1-931971-29-4. URL: `https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman`.

[153] Akshat Verma, Puneet Ahuja, and Anindya Neogi. “Power-aware Dynamic Placement of HPC Applications”. In: ICS ’08. Island of Kos, Greece, 2008, pp. 175–184. ISBN: 978-1-60558-158-3.

[154] Richard West et al. “Online Cache Modeling for Commodity Multicore Processors”. In: *SIGOPS Oper. Syst. Rev.* 44.4 (Dec. 2010), pp. 19–29. ISSN: 0163-5980.

[155] Tom White. *Hadoop: The Definitive Guide*. 1st. O’Reilly Media, Inc., 2009. ISBN: 0596521979, 9780596521974.

[156] Christian Widmer, Yasemin Altun, and Nora C. Toussaint. *Multitask Multiple Kernel Learning (MT-MKL)*. 2010.

[157] Alexander Wieder et al. “Orchestrating the Deployment of Computations in the Cloud with Conductor”. In: NSDI’12. San Jose, CA, 2012.

[158] Kishan Wimalawarne, Masashi Sugiyama, and Ryota Tomioka. “Multitask learning meets tensor factorization: task imputation via convex optimization”. In: *NIPS*. 2014.

[159] Ya Xue et al. “Multi-task learning for classification with Dirichlet process priors”. In: *JMLR* 8 (2007).

[160] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. “Wrangler: Predictable and Faster Jobs Using Fewer Resources”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC ’14. Seattle, WA, USA: ACM, 2014, 26:1–26:14.

[161] Neeraja J. Yadwadkar et al. “Multi-Task Learning for Straggler Avoiding Predictive Job Scheduling”. In: *Journal of Machine Learning Research* 17.106 (2016), pp. 1–37. URL: `http://jmlr.org/papers/v17/15-149.html`.

[162] Neeraja J. Yadwadkar et al. “Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC ’17. Santa Clara, California: ACM, 2017, pp. 452–465. ISBN: 978-1-4503-5028-0. DOI: `10.1145/3127479.3131614`. URL: `http://doi.acm.org/10.1145/3127479.3131614`.

[163] *Yahoo! Cloud Serving Benchmark*. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. 2017.

[164] *Yahoo! Cloud Serving Benchmark*. https://github.com/brianfrankcooper/YCSB/wiki/Implementing-New-Workloads. 2017.

[165] Ming Yuan and Yi Lin. "Model selection and estimation in regression with grouped variables". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68.1 (2006), pp. 49–67.

[166] Matei Zaharia. *The Hadoop Fair Scheduler*. http://developer.yahoo.net/blogs/hadoop/FairSharePres.ppt. 2012.

[167] Matei Zaharia. *The Hadoop Fair Scheduler*. http://developer.yahoo.net/blogs/hadoop/FairSharePres.ppt. 2012.

[168] Matei Zaharia et al. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling". In: *Proceedings of the 5th European conference on Computer systems*. EuroSys '10. 2010. ISBN: 978-1-60558-577-2. DOI: http://doi.acm.org/10.1145/1755913.1755940. URL: http://doi.acm.org/10.1145/1755913.1755940.

[169] Matei Zaharia et al. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling". In: EuroSys '10. Paris, France: ACM, 2010, pp. 265–278. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755940. URL: http://doi.acm.org/10.1145/1755913.1755940.

[170] Matei Zaharia et al. "Improving MapReduce Performance in Heterogeneous Environments". In: *OSDI*. 2008.

[171] Matei Zaharia et al. "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*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.

[172] Xiao Zhang et al. "Processor hardware counter statistics as a first-class system resource". In: *HOTOS'07*. San Diego, CA, 2007, pp. 1–6.

[173] Wei Zheng et al. "JustRunIt: Experiment-based Management of Virtualized Data Centers". In: USENIX'09. San Diego, California, 2009, pp. 18–18.

[174] Zhi-Hua Zhou and Xu-Ying Liu. "Training Cost-Sensitive Neural Networks with Methods Addressing the Class Imbalance Problem". In: *IEEE Trans. on Knowl. and Data Eng.* 18.1 (Jan. 2006), pp. 63–77. ISSN: 1041-4347. DOI: 10.1109/TKDE.2006.17. URL: http://dx.doi.org/10.1109/TKDE.2006.17.

[175] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". In: ASPLOS XV. Pittsburgh, Pennsylvania, USA, 2010, pp. 129–142. ISBN: 978-1-60558-839-1.