UNIVERSITY OF CALIFORNIA, MERCED

Scheduling, Characterization and Prediction of HPC Workloads for Distributed Computing Environments

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

in

Electrical Engineering and Computer Science

by

Mina Naghshnejad

Committee in charge:

Professor Mukesh Singhal, Chair
Professor Florin Rusu
Professor Sugjin Im

2019

Copyright Notice

The Dissertation of Mina Naghshnejad is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Florin Rusu

Sungjin Im

Mukesh Singhal, Chair

University of California, Merced

2019

Dedication

To all friends, family members, teachers, professors and book authors and influencers who taught me that there is a meaning in life and it worths to persist and pursue my ambitions.

Contents

List of Figures

List of Tables

List of Algorithms

Preface

`http://cloudlab.ucmerced.edu/~Mina`

- Naghshnejad, Mina. (2019). Scheduling, Characterization and Prediction of Workloads for High Performance Computing Systems. Ph.D. dissertation. University of California, Merced.

Acknowledgment

Similar to many others, Ph.D. was a long journey for me. Through this journey, I learned a lot, lost some of the best people, and found some of the best people. As an Iranian national, being able to focus on research for six years and not having to worry about other aspects of life was a great opportunity despite all the concerning matters happening to my closest friends and family in these six years.

I want to thank Professor Mukesh Singhal, who trusted my capabilities and accepted me as his Ph.D. student. Additionally, I would like to thank Professor. Sungjin Im, who introduced me to research on job scheduling. I also like to thank Professor Florin Rusu, who accepted to be part of my dissertation committee. Additionally, I would like to thank the graduate division, specifically Dean Marjorie Zatz and Professor Kello, who facilitated supporting learning and development programs that I benefited while I was doing my Ph.D.

Last but not least, I would like to thank my family: my parents who taught me how to strive for knowledge, my two sisters who supported me and encouraged me to pursue my goals, and my partner in life who taught me how to be precise and firm in expressing my ideas. It would not be possible without their help.

Curriculum Vita

Education

- **University of California, Merced**. Merced, CA, USA. (2013 – 2019).
  Ph.D. in Electrical Engineering and Computer Sciences.

- **Shahid Beheshti University**. Tehran, Iran. (2011 - 2013).
  M.Sc. in Computer Science.

- **Sharif University of Technology**. Tehran, Iran. (2005 - 2010).
  B.Sc. in Computer Science.

Research papers

Published papers

1. Handling prediction Inaccuracies in Backfilling Scheduling Approaches M Naghshnejad, M Singhal, Journal of Super Computing, 2019. Link
2. Naghshnejad, Mina, and Mukesh Singhal. "Adaptive Online Runtime Prediction to Improve HPC Applications Latency in Cloud." 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018. Link
3. Scheduling Jobs with Non-uniform Demands on Multiple Servers without Interruption S. Im, M. Naghshnejad, and M. Singhal, Proceedings of the IEEE INFOCOM 2016. Link

To be Submitted

4. Predicting Resource Consumption with LSTM Recurrent Neural Network M Naghshnejad, M Singhal
5. Predicting Job Runtimes with Deep Mixture Density Networks M Naghshnejad, M Singhal

Professional Work Experience

- Quantitative Research Associate, Wells Fargo Corporate Risk Modelling, San Francisco, CA, USA July 2019-Now
- Summer Machine Learning Research Intern, Qeexo, Pittsburgh, PA, USA Summer 2018
- Data Analysis Specialist, MTN Irancell, Tehran, Iran 2010-2011

Abstract

Scheduling, Characterization and Prediction of Workloads for Distributed Computing Environments
A Ph.D. dissertation by: **Mina Naghshnejad**
Electrical Engineering and Computer Science
University of California, Merced. 2019.
Committee chair: Professor Mukesh Singhal.

As High Performance Computing (HPC) has grown considerably and is expected to grow even more, effective resource management for distributed computing systems is motivated more than ever. As the computational workloads grow in quantity, it is becoming more crucial to apply efficient resource management and workload scheduling to use resources efficiently while keeping the computational performance reasonably good. The problem of efficiently scheduling workloads on resources while meeting performance standards is hard. Additionally, non-clairvoyance of job dimensions makes resource management even harder in real-world scenarios. Our research methodology investigates the scheduling problem compliant for HPC and researches the challenges for deploying the scheduling in real world-scenarios using state of the art machine learning and data science techniques.

To this end, this Ph.D. dissertation makes the following core contributions: a) We perform a theoretical analysis of space-sharing, non-preemptive scheduling: we studied this scheduling problem and proposed scheduling algorithms with polynomial computation time. We also proved constant upper-bounds for the performance of these algorithms. b) We studied the sensitivity of scheduling algorithms to the accuracy of runtime and devised a meta-learning approach to estimate prediction accuracy for newly submitted jobs to the HPC system. c) We studied the runtime prediction problem for HPC applications. For this purpose, we studied the distribution of available public workloads and proposed two different solutions that can predict multi-modal distributions: switching state-space models and Mixture Density Networks. d) We studied the effectiveness of recent recurrent neural network models for CPU usage trace prediction for individual VM traces as well as aggregate CPU usage traces. In this dissertation, we explore solutions to improve the performance of scheduling workloads on distributed systems.

We begin by looking at the problem from the theoretical perspective. Modeling the problem mathematically, we first propose a scheduling algorithm that finds a constant approximation of the optimal solution for the problem in polynomial time. We prove that the performance of the algorithm (average completion time is the constant approximation of the performance of the optimal scheduling. We next look at the problem in real-world scenarios. Considering High-Performance Computing (HPC) workload computing environments as the most similar real-world equivalent of our mathematical model, we explore the problem of predicting application runtime. We propose an algorithm to handle the existing uncertainties in the real world and show-

case our algorithm with demonstrative effectiveness in terms of response time and resource utilization. After looking at the uncertainty problem, we focus on trying to improve the accuracy of existing prediction approaches for HPC application runtime. We propose two solutions, one based on Kalman filters and one based on deep density mixture networks. We showcase the effectiveness of our prediction approaches by comparing with previous prediction approaches in terms of prediction accuracy and impact on improving scheduling performance. In the end, we focus on predicting resource usage for individual applications during their execution. We explore the application of recurrent neural networks for predicting resource usage of applications deployed on individual virtual machines. To validate our proposed models and solutions, we performed extensive trace-driven simulation and measured the effectiveness of our approaches.

Chapter 1

Introduction

High Performance Computing is behind most of recent advances in science and engineering. Researches performed in national labs are not possible without the enormous parallel interconnected super computers known as high performance systems. Research in scientific fields such as computational physics, computational chemistry, bio-informatics rely on HPC systems to perform complex large scale computational experiments.

As the demand for HPC is growing, more advanced resource management and application schedulers are required to provide desired performance demands of customers while keeping the expenses of running HPC systems reasonable.

1.1 Research Motivation and Objectives

As computational requirements are increasing, the usage of public and private distributed systems for High Performance Computing is increasing. These distributed systems offer significant benefit to companies in various industries by relieving them from the necessity in setting up basic hardware and software infrastructures, and thus enabling them to focus more on innovation and creating business value for their services. Moreover, individual developers with innovative ideas for new Internet services no longer require significant capital outlays in hardware to deploy their service or staffing expenses to operate it.

With the increase in the usage of distributed systems and the fact that the semiconductor industry is not expecting further exponential growth of computational power per chip area, the role of effective resource management is more apparent than ever (15). Resource management is an essential part of cloud computing administration. Resource management is the process of allocating resources to a set of applications in a manner that seeks jointly meeting the performance objective of applications and infrastructure providers. Resource management in distributed systems is a difficult problem, due to the large scale of modern data centers, the heterogeneity of resource types and their inter-dependencies, the variability and unpredictability of the response time and resource usage, as well as the wide spectrum of objectives of

the different actors in a cloud ecosystem. Consequently, both academia and industry have started significant research efforts in this area (76).

Most of the problems that arise in resource management problems in distributed systems are NP-complete. Cloud providers and brokers need to avoid high overheads for management modules. Thus, designing fast near-optimal algorithms to handle resource management is of great concern.

Another issue in real-world scenarios of scheduling applications is the uncertainty about the dimensions of incoming applications. Most of the times, neither the runtime of the application nor its resource usage is known to the distributed system vendors. For that reason, designing prediction approaches that can predict the application runtime and resource usage in an online manner is highly demanded. Using cloud resources extravagantly is a double-edged sword because performance improvement comes with the expense of losing resource efficiency. Designing a resource manager that meets resource efficiency as well as performance leads to significant revenue for the cloud provider and revolutionizes the cloud computing business.

In this Ph.D. thesis, we focus on High Performance Computing systems, and we mostly focus on scientific computing workload traces from national labs as our datasets. Only in the last chapter, we use the subset of long-running traces from Microsoft Azure. The overall goal of this Ph.D. thesis is to propose solutions for improving resource management and scheduling workloads on distributed computing systems. In part of our work, we propose approximation algorithms to schedule HPC applications with a constant approximation of optimal average completion time of applications.

In addition to constant approximation solution for the case of known application runtime, we also study the scenarios where application runtimes are unknown. We propose a solution to handle inaccuracies for application runtimes. We propose two different approaches for predicting application runtimes by using the past traces of applications. Additionally, we study the resource usage pattern of virtual machines and propose a deep recurrent neural network for predicting individual virtual machine workload usage pattern.

1.2 Research Challenges

Creating an optimal scheduling plan even when the runtime and resource requirement of applications is known is an NP-Hard problem. One challenge is to find sub-optimal solutions with polynomial computation time. Cloud and cluster vendors rent computational resources to entities. Most of these entities do not want the cloud/cluster vendor to know the contents of the applications they run on third party distributed systems. On the other hand, the distributed system vendor needs to manage its resources to provide high-performance service while using the computational/memory/network services efficiently to optimize its revenue.

For practically optimizing the planning of applications of customers, the vendor needs to have a near-accurate prediction of the runtime and resource usage of the

applications. However, the vendor does not have much detail about applications submitted by the customers. In fact, in most of the scenarios, the actual runtime of applications is unknown. Consumers are often asked to input an estimate of their application runtime, but studies show that these estimates are far from accurate.

Predicting the runtime of applications is not easy. HPC applications are becoming more diverse, and their characterization and prediction is getting more difficult. Prediction approaches should have low overhead and should be deployable in an online manner so that the predicted runtime is used for scheduling purposes. Another challenge for predicting runtimes is that often privacy considerations do not allow the vendor to interrogate the codes of submitted applications. The privacy considerations and legislations imply that prediction approaches should work with a minimal set of features available from application trace.

Resource usage prediction is a hard problem. In fact, the resource usage of individual applications is hard because of the existing noise and also because of the large variance of resource usage among different applications.

1.3 Research Questions and Contributions

An important research question is whether we can predict the application runtimes with minimal information we have from the application submission logs. Another important question is if there is an unavoidable uncertainty in runtimes of applications, can we still devise scheduling algorithms to improve performance? Another critical research question is if we can predict the individual resource usage patterns of applications using the past resource usage of applications.

With the advent of dynamic virtual machines and container technology and the feasibility of creating containers for each application, there is the opportunity of consolidating resource allocation and application schedule. Our work lies in the intersection of resource allocation in the cloud and workload prediction for distributed systems. To answer the current concerns about resource efficiency in cloud and regarding the recent popularity of actualization methods, we consider the problem of scheduling virtual machines to optimize performance. To keep the resource usage efficient, we hard constrain the usage of resources and do not run an application unless the required resource for the application is available on at least one of the servers in the system. For performance optimization, we consider the objective of minimizing the total completion time of applications as it is one of the most popular objectives for performance optimization and we are able to prove upper bounds for this objective for our proposed algorithms.

We propose two different approaches to the runtime prediction of HPC applications. One uses state-space models, and the other applies deep mixture density networks for prediction the runtime of HPC applications on distributed systems. We also propose the usage of the recurrent neural network for predicting resource usage for each individual virtual machine deployed in the distributed system.

specific contributions of this Ph.D. thesis include:

1. design non-preemptive offline scheduling solutions for HPC workloads
2. design approaches to predict the runtime of HPC applications in the cluster using past traces using machine learning
3. design scheduling solutions to handle runtime inaccuracies of HPC workloads
4. predicting individual resource usage of virtual machines deployed in distributed systems

1.4 Thesis Outline

Chapter 2 provides an overview of high performance computing, scheduling methodologies for high performance systems and approaches for predicting runtime and CPU usage of applications in high performance computing systems. The next five chapters are our contributions. In Chapter 3.5, our proposed scheduling algorithms for non-preemptive applications is presented. In Chapter 4, a novel methodology for the handling of runtime inaccuracies in HPC clusters is presented. Two novel prediction approaches for HPC application runtimes are presented in Chapters 5 and 6. Finally, a novel approach for predicting CPU usage of applications is presented in Chapter 7. We conclude the thesis in Chapter 8.

Chapter 2

Background

In this chapter, we present required background about well-established theories, methodologies, and technologies which are used in this thesis. In this regard, we first define High Performance Computing; we then explore scheduling and resource management practices for high performance scheduling. We then discuss the necessity of performance prediction for scheduling and resource management of HPC applications. After that, we present the problem of runtime prediction and resource prediction for HPC applications and explore existing approaches.

2.1 High Performance Computing

High Performance Computing, also known as HPC, is named after the execution of applications in fields including science, engineering, health, and finance. The main characteristic of these applications is high computational loads and large input and outputs. A computing system should be defined as High Performance Computing if it supports the execution of large-scale, performance-oriented applications at the smallest cost, with the shortest possible runtime, within some time constraint.

Definition 1 (High Performance Computing)**.** Execution of applications in interconnected clusters of computational servers with specific performance gourantee is called High Performance Computing

Definition 2 (HPC applications)**.** Large scale applications with large input and/or outputs that require specific performance standards to be met.

Definition 3 (HPC systems)**.** HPC systems support the execution of HPC applications. Specifically, they provide parallel processing, resource sharing, and timesharing.

Most critical costs for running and maintaining HPC clusters are cooling and power provision. The facilitation of the desired performance while optimizing costs requires careful scheduling at job level and process level and effective resource management.

For the majority of this thesis, we will focus on HPC systems for scientific computing, but most of the concepts can be considered in the more general HPC system systems as well.

2.2 Scheduling and Resource Management in HPC Systems

This thesis studies scheduling of HPC applications. HPC applications are traditionally applied in private clusters. Recently, with increasing popularity of Cloud systems, HPC as a Service is introduced and provided by multiple Cloud providers. HPC as a Service provides high performance virtual environments in a pay as you go basis to Cloud customers. Both in private clusters or in HPC as a Service Cloud environments, it is mandatory to carefully manage resources to optimize service to the customers and keep the cluster/cloud service sustainable. In both services, customers require specific level level of perfomance for their applications. At the other hand, Cloud/-cluster vendor needs to minimize resource usage of the distributed system to optimize its revenue (in case of virtual environments in Cloud) or make cluster maintenance affordable (in case of private clusters.

2.2.1 Scheduling in Distributed Systems

"In computing, scheduling is the method by which work specified by some means is assigned to resources that complete the work "(Wikipedia). The work may be virtual computation elements such as processes which are in turn scheduled onto hardware resources such as processors. Early scheduling algorithms were taken from the field of operations management and applied to computers. This reality should be no surprise that assembly lines also require scheduling, and many of the same concerns exist therein, including a laser-like desire for efficiency.

The scheduling problem can be considered offline- where all jobs are given before starting the scheduling algorithm - or online (dynamic)- where jobs are submitted during the scheduling and scheduler needs to decide how to schedule the job before next job is submitted. In this thesis, we will discuss the scheduling method focused on parallel jobs.

First, we will elaborate on how the distributed system runs applications across the system. We then explain the role of the scheduler. In general, job scheduling involves two inter-dependent steps: the allocation of jobs to processors (space-sharing) and determining the start time of each job (time-sharing). When a job is submitted to the system, the processor to run the job cooperatively is determined. This is called job allocation, and it involves space-sharing. When the job is submitted to the system, multiple attributes of the job is also submitted that quantifies the resource usage and expected runtime. It is important to note that the system always maintains an information table, containing information about current running jobs and the status

of resources being used. When a job is submitted, the system manager finds the most suitable processor to meet the requirement of the job. This job is then assigned to the processor, and system tables are updated accordingly.

The most famous scheduling algorithms include First In First Served (FCFS) that processes jobs in the order of their submission. It is proved that in the model where only one job at a time can be processed by each server, the Shortest Job First (SJF) algorithm is optimal for homogeneous servers. However, in our problem setting- which we discussed in Chapter 1- multiple jobs can run concurrently on available resources scheduling jobs is an NP-Hard problem even in the case of a single server.

Desired Properties of an Ideal Scheduler for HPC Applications

Regarding the business requirement and usage of HPC clusters, the following properties are expected from HPC schedulers (94):

Efficiency: it has two meanings: one is that it should improve the performance of scheduled jobs as much as possible; the other is that the scheduling should incur reasonably low overhead so that it won't counter attack the benefits.

Fairness: sharing resources among users raises new challenges in guaranteeing that each user obtains his/her fair share when demand is heavy. In a distributed system, this problem could be exacerbated such that one user consumes the entire system. There are many mature strategies to achieve fairness on a single node.

Dynamicity: the algorithms employed to decide where to process a task should respond to load changes, and exploit the full extent of the resources available.

Transparency: the behavior and result of the execution of a task should not be affected by the host(s) on which it executes. More specifically, there should be no difference between local and remote execution. No user effort should be required in deciding where to execute a task or in initiating remote execution; a user should not even be aware of remote processing, except maybe better performance. Further, the applications should not be changed greatly. It is undesirable to have to modify the application programs in order to execute them in the system.

The general problem of optimally mapping tasks to machines in an HPC suite has been shown to be NP-complete. Different heuristics are developed to perform this mapping. The problem gets even harder to solve when we running multiple jobs on single server concurrently as long as enough resources are available. In the following subsection, we explore common approaches for scheduling HPC applications.

2.2.2 Scheduling HPC applications in private clusters

The HPC applications are submitted to a central Application Management System. The job management system decides the allocation of resources to the applications. As the resources are finite, the applications may need to wait until they acquire resources. The users are asked to provide running time and the required CPU and memory for their applications. It is well known that the users submit an overestimated

runtime mostly because the application manager kills the applications if they take longer than the user estimated duration (71).

In order to better present the problem, we first define our notion of a job in HPC cluster: a *job* or *application* j is considered with specific submission time and resource requirement. At the time of submission, a value of runtime and resource requirement is submitted by the user. There are n independent jobs (indexed by integers), where application j has the following characteristic:

- Submission time: r_j
- Resource requirement: d_j
- Actual running time: p_j
- Requested running time: $\hat{p}_j$.
- Additional features (descriptors) including the user that submitted the application, the time of the day the application submitted, etc.

In Fig 4.1, an application j is illustrated as a solid rectangle, with length equal to actual runtime (p_j) and resource requirement equal to d_j. In the right figure, the abstraction of resources in the HPC cluster is illustrated. The vertical dimension represents the total resources in the system, and the horizontal axis denotes time. In this section, we first present a background on scheduling approaches for HPC clusters and introduce the issue of uncertainty in HPC workload traces. We then present the formulation of the problem we study in this paper.

Common Scheduling Algorithms for HPC Workloads

FCFS (First Come First Served) is the most well-known scheduling algorithms for HPC jobs. FCFS schedules jobs in order of their submission. FCFS is a list scheduling algorithm that prioritizes jobs based on their submission time. In list scheduling algorithms, also known as queue-based scheduling algorithms, if there are enough available resources, resources are allocated to the submitted job, and the job starts to process. Otherwise, the job is kept in a queue. Using FCFS does not consider the geometry of jobs to pack them tightly into resources. To improve the performance of FCFS, two strategies have been proposed: Backfilling (FCFS-BF) (135) and Plan-Based scheduling (160) algorithms. In Fig 4.2, an FCFS-Backfilling scheduling strategy is compared with a plan-based scheduling algorithm. While Backfilling backfills the free resources available after FCFS scheduling with smaller jobs from the back of the queue, plan-based scheduling uses the runtime of jobs in the queue to find the near-optimal ordering jobs before assigning the jobs. As these two groups of scheduling algorithms are the building blocks of our hybrid scheduling platform, we elaborate their characteristics in the following subsections.

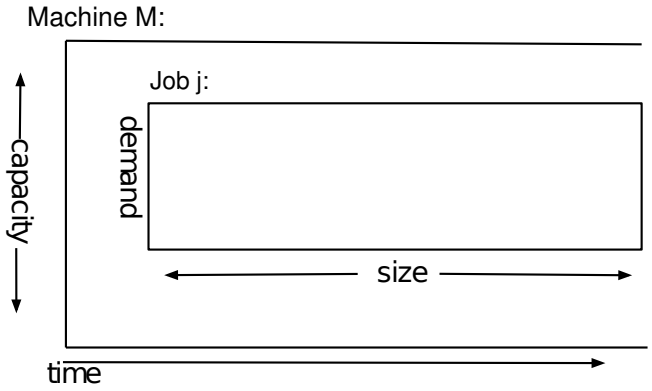

Figure 2.1: Each job j is illustrated as a two-dimensional rectangle with height equal to its demand d_j and width equal to its size p_j. Each server has the unit capacity. Jobs can run simultaneously on each server as long as the total demand/height of running jobs do not exceed the server's capacity.

FCFS Scheduling with Backfilling

In FCFS with backfilling, jobs are prioritized based on their submission time to the system. In FCFS with backfilling, a rule is used to select some jobs from the back of the waiting queue to run earlier. Several variety of backfilling algorithms are proposed including easy (128), conservative (135) and slack backfilling (138) algorithms. Although FCFS only rely on submission time to order the jobs, all these backfilling approaches rely on runtime values to make the backfilling decision. Most of the resource management systems deployed in HPC clusters including SLURM (154), Cobalt (38), IBM LoadLeveler (128) use FCFS with backfilling. FCFS scheduling algorithms are used mainly due to their simplicity and scalability as well as stability to inaccurate input runtimes. The easy-backfilling algorithm acts like a greedy first fit scheduler in the case that the next job in the queue has more resource demand than the available resources. It takes the first job from the back of the queue that fits into the available space.

One important observation is that as EASY-backfilling tries to backfill jobs greedily into available holes created by the FCFS ordering of jobs, its performance does not degrade substantially with inaccurate runtime estimates. On the other hand, the performance of the EASY does not improve substantially with more accurate runtime values. FCFS-SJF was proposed in (142) to use the application runtime for backfilling decision. In FCFS-SJF, the backfilled jobs are chosen in the order of increasing runtime. FCFS-SJF is commonly used in the works that propose more accurate prediction approaches to runtime prediction as SJF-BF is more sensitive to runtime prediction accuracy than EASY. We will also consider FCFS-SVF that orders jobs based on volume(multiplication of runtime and required CPU). FCFS-SJF and FCFS-SVF have some level of sensitivity to runtime accuracy, but still, have

acceptable performance in the absence of accurate runtime prediction. In our experiments, we use FCFS-SJF and FCFS-SVF as representatives of FCFS with backfilling algorithms.

Plan-based Scheduling Algorithms

On the other side of the scheduling algorithms spectrum are the plan-based algorithms. Instead of deploying jobs immediately, plan-based approaches make a scheduling plan-based on a group of submitted jobs. They try to find a near-optimal ordering of jobs to optimize scheduling performance. The main issue with plan-based scheduling algorithms is the fact that their performance is highly sensitive to the accuracy of jobs' runtime predictions. As in real-world scenarios, user runtime estimates used for scheduling are not accurate; plan-based scheduling algorithms do not perform well. We study several plan-based and backfilling approaches and propose an adaptive hybrid scheduling platform. Our sensitivity analysis experiments in the next subsection show how plan-based work well with accurate and backfilling with inaccurate predictions. Plan-based scheduling algorithms try to search over the Solution space to make the best scheduling decision for each job. As the problem is dynamic and jobs are submitted over time, the planning routine needs to be done periodically and based on the jobs already in the systems. Several plan-based scheduling policies have been proposed. Some policies propose a complete search over the solution space, and some propose local search to improve the computation overhead (160; 152; 82).

Although these methods claim to have better performance than backfilling scheduling methods, several issues make them less popular for cluster managers. First, for most of these approaches, the computational the overhead for decision making makes them less favorable. Furthermore, they completely rely on job runtimes for their decisions and perform poorly if the runtimes are not accurate. We will discuss this issue in the next section.

2.2.3 HPC as a service in the Cloud

Many pieces of research have found scientific workflows, referred to as workflows in this thesis, an attractive model of building large scale applications for heterogeneous parallel and distributed computing systems. Typically, a scientific workflow application consists of several (legacy) programs in the form of a dependency graph, where the input of some of these algorithms may depend on the output of others. In scientific computing, usually users that submit the jobs tell the scheduler an upper bound on how long they will run, and the scheduler can make a plan for when to launch each job(offline scheduling). These scientific workflows are run in private clouds/clusters or public clouds like commercial Infrastructure as service environments, including Amazon Elastic Computing Cloud (EC2) and Google Cloud Platform (GCP). To deploy workflow scheduling in current cloud systems, the user has to choose between different available resource types and specify how long he wants to rent the resource. So the resource requests made by customer consist of resource demand and duration

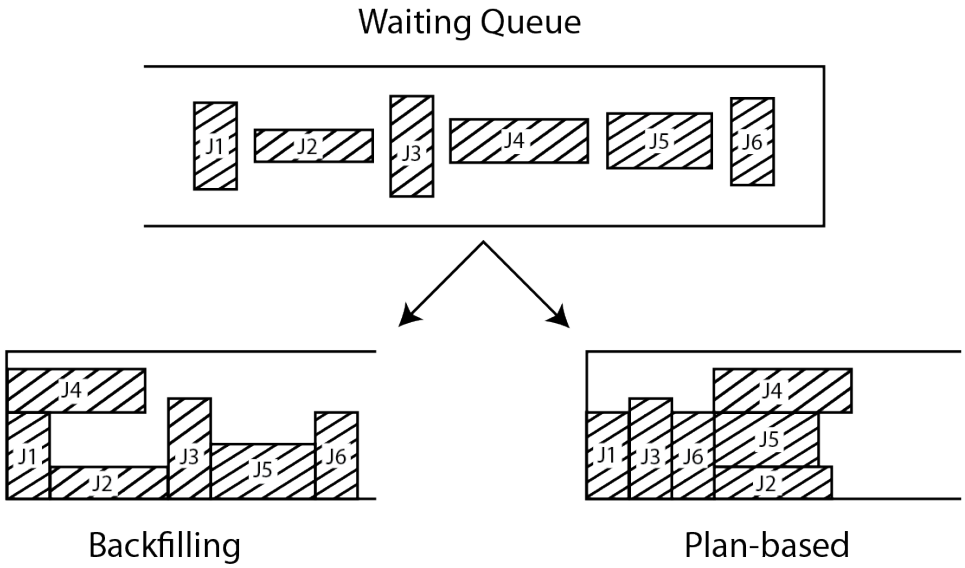

Figure 2.2: The comparison of a plan-based scheduling algorithm (online-SJF) and a backfilling scheduling algorithm (FCFS-SJF) algorithms on an example of seven jobs.

of time that the resource needs to be used. This framework is similar to our modeling except that we assume each workflow as a single solid application.

2.2.4 Resource management in distributed systems

Since the development of cloud systems, resource management has been an indispensable part of protocol design and architecture. However, the early implemented algorithms are too simple and do not consider all objectives in scheduling applications on the available resource.

With the advance of computer and internet technologies, paradigms of cloud computing have been successfully used on several information systems in recent years. Although cloud computing systems nowadays provide a better way to carry out the submitted tasks in term of responsiveness, scalability, and flexibility, The current methods are far from optimal. Some methods only aim to achieve fairness among users, while others try to optimize responsiveness. However, to our knowledge, there has not been so many works on optimizing performance while keeping resource usage efficient. One reason for this is the fact that scheduling can have many different and contradictory aims that it is not possible to satisfy all of them with one single scheduler. Different role-players in the Cloud have different goals to achieve. Traditionally resource scheduling in the Cloud is handled in different levels:

1. Application layer: applications, user tasks, workflows

2. Virtualization(Platform layer): virtual machines, databases, integrations, secu-

rity

3. Deployment(Infrastructure) Layer: Networking, Storage, server hardware, services.

Each of these levels has its own objectives for sharing the resources among the applications. End-user is seeking fairness and responsiveness. Platform layer is looking for quality of service to satisfy end-user as well as the reliability of the service. It may also seek to optimize its resource usage if it is hiring servers from a cloud provider. The infrastructure layer is looking to optimize power consumption, hardware usage. Based on the objectives we consider for resource management, different algorithms and methods have been designed and implemented.

Resource allocation in distributed systems

In slot based resource management systems, resource allocation is handled by the Cloud provider to allocate virtual machines on physical machines. Initially, the resources are divided among virtual machines or containers in the infrastructure level. Tasks belonging to applications are scheduled on virtual machines in platform level. From the methods and algorithms for resource allocation in the Cloud, we want to highlight bin packing based algorithms. Bin packing based algorithms have been proposed to allocate virtual machines(VMs) on servers. In these problems, server resources such as CPU, memory, bandwidth, and storage are dimensions of the packing problem. These algorithms are often used in server selection phase to achieve cost minimization, energy efficiency, and utility maximization. For the objective of minimizing the number of servers, (147) and (23) proposed approximation algorithms assuming that servers are homogeneous. (95) applied a multidimensional vector packing algorithm to allocate virtual machines in a cloud system. In some works, including (132) virtual migration facilitated adaptive approximation algorithms. However, in all these bin packing algorithms, only the objectives of the cloud provider are taken into account, and customer SLA satisfaction regarding responsiveness is not taken into consideration.

Modern models of resource allocation take advantage of creating and destroying virtual containers on the go. In these models, end-users submit their requests with the exact amount of resources that they require, and the cloud manager will be responsible for providing the requested resources. One issue in resource allocation is the concept of fairness among users or applications. In order to allocate resources in a fair manner, max-min sharing, which maximizes the minimum allocation received by a user in the system is employed and all the users whose demand cannot be satisfied share the remaining resource. Dominant resource fairness(148) is proposed to achieve fairness in the case of heterogeneous resource demands. However, in all these resource allocation algorithms, the applications are executed even if the sufficient resource is not available. In a more constrained model, applications are executed only if there is sufficient resources for their deployment, and otherwise they will be waitlisted.

Task Scheduling in Application Layer of the Cluster

Most of the existing scheduling algorithms take care of the scheduling in the platform level. In these models, the scheduler is given a coarse-grained slot of resources to share among the submitted applications with some criteria. The most famous and simplest scheduler for cloud systems is FIFO, where jobs are executed in order of their submission. FCFS is the simplest member of a group of schedulers that can be categorized under the name of list scheduling algorithms. The jobs are loaded into a queue and scheduled to execute according to their order in the queue. More advanced list scheduling algorithms are designed and released after FCFS. Later on, the ability to set the priority of a Job was added. Facebook and Yahoo contributed significant work in developing schedulers, i.e., Fair Scheduler (44) and Capacity Scheduler (Im et al.) respectively, which subsequently released to Hadoop Community. In capacity scheduling tasks for each user form a queue. Each queue has a configurable portion of the resources. In fair scheduling, Jobs are allocated to pools. The scheduler guarantees that each pool gets a minimum portion of resources over time using preemption. Delay Scheduler (90), Dynamic Proportional Scheduler (122) offer differentiated service for Hadoop jobs allowing users to adjust the priority levels assigned to their jobs. There has been extensive work on improving these methods. Deadline Constraint Scheduler (78) addresses the issue of deadlines but focuses more on increasing system utilization.

The Schedulers described above attempt to allocate capacity fairly among users and jobs; they make no attempt to consider resource availability on a more fine-grained basis. More specifically, they do not consider the heterogeneity of resource demands by different applications. Another concern is also the need to configuring slots for heterogeneous clusters of servers, as slots by default have different dimensions on heterogeneous servers. These algorithms are mainly aiming to achieve the quality of service based on SLA agreements with customers, including fairness and service responsiveness and ignore resource efficiency.

Virtual Machine Scheduling in Clusters

As most of the scheduling algorithms in the Cloud do not consider optimizing resource usage as an objective or constraint, considerable waste of resources happens in the current management of cloud systems. About forty-five percent of the cost in cloud computing systems is caused by physical nodes (64).

There have been several works in scientific workflow scheduling to target performance optimization while keeping resource usage efficient. One approach to make efficient use of resources is considering resource usage as one of the objectives in addition to performance objectives. In (90), a pareto based multi-objective workflow scheduling with two objectives of makespan and cloud cost based on the EC2 cloud model is proposed. The problem with the proposed algorithm is that the greedy algorithm is time-consuming and imposes an overhead if applied in large scale cloud computing systems.

Another solution is post-processing of output workflow schedules. In (44), a new algorithm Maximum Effective Reduction(MER) is proposed. MER is a post-processing resource efficiency algorithm that optimizes resource usage of a workflow schedule. It trades off a makespan increase for reducing maximal resource usage by consolidating tasks with the exploitation of resource efficiency in the original workflow schedule.

In a theoretical paper, (49) several online algorithms are proposed to minimize the flow time of the jobs; meanwhile, the total resource used should not exceed that of the server. Unfortunately, no experiments or simulations are included in that work to prove the superiority of their work in practice. Another issue about their work is the fact that they assume that jobs can be preempted free of charge, which is not a realistic assumption. In (Im et al.), we considered the same model but did not allow preemption to make the model more realistic. We proposed several efficient and easy to implement algorithms to minimize total completion time, and simulations confirmed the good performance of our algorithms in practice.

2.2.5 Existing Challenges for Scheduling HPC Applications

As current resource allocation solutions are not considering performance optimization objectives and task scheduling algorithms are not considering resource efficiency, we aim to propose resource management systems that accommodate both concerns. The few resource-efficient scheduling algorithms proposed so far for scientific workflow scheduling are time-consuming and not suitable for deploying in public clouds. In our model, we consider performance optimization as objective, and hard constrain the resource usage by requiring that at any time the sum of used resources by running applications should not exceed the resources on any of the servers.

The recent improvements in virtualization technologies allow us to allocate each application on a single virtual machine or container. That allows us to consider the problem of virtual machine scheduling where each virtual machine is supposed to run if and only if the resources required by the related application is available on one of the resources in the Cloud (33).

Another issue is preemption. Most of the current algorithms assume that the preemption of virtual machines can be done free of charge. However, this assumption is not valid for HPC systems. Preemption may be costly due to context switching overheads or may not be allowed due to system restrictions or security considerations (108). For this reason, it is crucial to design algorithms for non-preemptive scheduling, although non-preemptive scheduling is a hard problem. These algorithms can be applied to schedule a subset of virtual machines or all of them based on the problem sets. That is why we have a specific emphasis on scheduling problem in non-preemptive settings. Several papers that discuss HPC scheduling and backfilling strategies consider the same assumption of non-preemption.

Another concern is that scheduling algorithms need to know the dimension of applications in order to schedule their execution into the available resources better.

When users submit HPC applications, they are asked to submit an estimated runtime. However, studies show that these assumptions are not accurate at all. One issue is that the cluster/Cloud management tell the users that they will kill the application if the application is not completed by after the estimated runtime. This encourages the users to submit a pessimistic runtime for their application. These upper-bound inaccurate runtimes are not helpful for optimal scheduling of application. In the next section, we will discuss approaches to predict runtime and resource usage of the application. The purpose of these predictions is to improve scheduling performance.

2.2.6 Research Methodology for Scheduling and Resource Management of Distributed Systems

Research in HPC scheduling is challenging mainly because complete information about real-world HPC workloads is not available. Private clusters, as well as third party Cloud vendors, keep most of the information regarding HPC traces and workload management information proprietary. The existing research in HPC scheduling can be classified into three methodologies:

1. Theoretical Analysis: In this methodology, lower bound and upper bound performance of the proposed algorithms are given by mathematically modeling the distributed system using the approximation algorithm.

2. Trace-based Simulation: In this approach, a real world system is simulated by using a simulated resource management system on simulated jobs. Simulated jobs can be synthetic workloads or trace logs of real-world distributed systems. Simulation enables repeatability of large scale experiments that provide data for analysis of the performance of algorithms.

3. Real System Experiments: This methodology provides a clear measure of the algorithm behavior but requires dedicated distributed system nodes and intensive time and investment on hardware. Also, a single experiment is not enough for reasoning about a general hypothesis. Also, workload conditions are hard to control in real systems, and this makes it hard to test specific hypothesis or algorithm settings.

A common practice to perform research on scheduling is to use real-world traces for simulating the cluster. In order to do so, multiple simulators have been proposed to mimic the distributed system. The parallel processing uses the information on arrival time, runtime, and resource usage to simulate running the applications on the cluster simulator. After running the applications, the performance metrics, including runtime, resource utilization, and bounded slow-down, is recorded for analysis and comparison between different scheduling algorithms.

In this thesis, we used the first two approaches. In Chapter 3.5, we propose a scheduling algorithm that runs in polynomial time, and we prove it is a constant approximation of the optimal answer. To prove this claim, we applied combinatorial

approximation. In the same chapter, we used Traced-based simulation to show the performance of the algorithm in practice. In Chapter 3.5, we performed both synthetic trace and real-world traces to simulate the algorithm on a computing cluster. In Chapter 4 and 5, we used trace-based simulation on real-world traces to test the impact of using predicted application runtime on scheduling performance.

2.3 Prediction Approaches for HPC Applications

Recently, machine learning approaches have been used for performance prediction for distributed systems. A group of these approaches predicts the performance metric values directly from data available about each application. Another group, first predict performance-related features, including runtime and resource usage. These features are, in fact, the requirement of the application to reach desirable performance. They use these predictions to perform simulations and calculate the values of performance metrics. Performance prediction helps to benchmark different scheduling approaches. In other hands, the need for workload characterization and prediction arise from the insufficient information at the time of submission of applications. Many scheduling and resource management algorithms require an accurate value of runtime and resource usage to reach to desired performance and efficiency results. As these values are not available, machine learning can be used to provide estimations.

Existing approaches for predicting runtime and resource requirement applications fall into two main groups:

1. White-box prediction approaches: In these approaches, the prediction methodology has a clear view of the application, including its code, arguments, preprocessing, and expected output. Several prediction-based scheduling approaches, including (19), and (37) perform application profiling to gather information about the current application. However, these approaches are not practical for large-scale deployment. There are several works including (111) on interference detection of applications which is not applicable to our problem setting (single application per virtual machine).

2. Black-box prediction approaches: In these approaches, the prediction methodology only has the trace log of the application. The trace log usually includes the user who submitted the application, estimated runtime by the user, requested resources. In addition, some system information is also available to the prediction model, including currently running applications from the same user, the current state of the resource usage in the system, etc.

In real-world scenarios, most often, the cluster management is not allowed to look into details of the applications. For this reason, we only consider black-box prediction approaches in this thesis.

2.3.1 Runtime Prediction for Applications in HPC Clusters

The necessity of runtime prediction for parallel applications has been highlighted since the last years of 20th century (58). Different approaches have been proposed to predict HPC application runtimes with different machine learning methodology as well as prediction features inputs (142; 139). In backfilling algorithms, the runtime of jobs is needed to determine when the processors of the currently running jobs will be available and also in determining the eligibility of the waiting job from the back of the queue to be executed (142).

Empirical studies of traces from HPC clustering sites show that user estimates are generally inaccurate (104). For this reason, there have been several works in trying to calculate better estimations of application runtimes. As users of HPC clusters tend to repeat similar applications, it is conceivable that historical logs of previous runs can be used to predict the runtime of future applications.

In statistical approaches, a distribution is pre-assumed for the data points, and the prediction is performed by inputting the characteristic of the new input data to the distribution. Some approaches consider multiple possible distributions, and after determining which distribution the new job fits into, they calculate the prediction of the response time of the job. The process of finding the parameters $a_1, a_2, ..., a_n$ and b is called training a machine learning model. When we have a model, we can use the model to predict the target value for future independent variables.

In another group of statistical approaches for job runtime prediction, job runtime values are considered as time series. The value of the time series at time t_i is the runtime that starts running at time t_i. One observation is that, in the created time series, time steps are not necessarily equidistance. Older time series based approaches used for job runtime prediction includes calculating the mean of the runtimes of the last two jobs from the same user or exponential smoothing which is a weighted mean of the previous jobs that discounts the older jobs and puts more weight on the runtime of the most recent jobs.

Several time-series based methodologies are proposed to use data from previous jobs to predict the runtime for newly submitted applications. They are mainly exponential smoothing and moving-average methodologies that predict future values based on the recent runtime values. As noted by (133), these methods are not accurate and will not improve scheduling performance and utilization significantly. To improve the accuracy of runtime prediction, we have proposed a time-series based approach in Chapter 5.

Another set of prediction approaches, find a mapping from a set of independent variables including time of submission of the job, user, estimated runtime, etc to the actual runtime of the job using machine learning techniques. We will next discuss these approaches.

Machine Learning for Predicting Runtimes

Some other researchers have considered additional features for each job in trace and perform prediction based on the similarity between these features of the jobs (102; 125). Some more recent works focus on the specific family of scientific workflows and use machine learning for predicting runtime and resource usage for these applications (100). Several works have proposed interrogating the codes to extract features for runtime prediction. This is also not practical in many cases due to privacy considerations.

The majority of online time series based prediction methods use simple forecasting rules including mean, moving average, and exponential smoothing. Sonmez et al. partition jobs into jobs submitted by the same user or jobs running at the same site and applies simple time series methods to predict the subsequent runtime of jobs based on the recent history. They consider the running mean of the last two jobs as the prediction method. Although these methods are easy to implement and do not need a large training pool, they are not very accurate. To configure the model based on the individual user and use the recently completed jobs in the same trace to strengthen the predicting power of the model, several online methods have been proposed (133; 139). In fact, the scarcity of relevant training data makes the model building cumbersome.

Supervised machine learning approaches provide tools for predicting a continuous dependent variable based on possibly multiple independent variables. In these approaches, a mapping is often trained from input data $X - i$ to y_is. For instance, in linear regression, target variable y is considered to be a linear function of independent variables $X - 1, .., X_n$:

$$y_i = a_1 X_{1i} + a_2 X_{2i} + ... + a_n X_{ni} + b \tag{2.1}$$

$a_1, a_2, ..., a_n$ and b are parameters of the linear regression model. One approach to finding the best mapping from X_i to y_i is to find the line with the minimum least-squares of error from the sample points. Linear regression is based on the assumption that independent variables follow Gaussian distribution. Studying the runtime data from HPC clusters show that the independent variables and target variable(runtime) is multi-modal. This implies that rather than assuming a single distribution, we need to consider a mixture of multiple Gaussians. In Chapter 6, we propose an approach based on deep mixture models to predict the runtime of applications. A group of supervised machine learning approaches function in an online manner. That means the objective function above is used to predict one individual job and it is retrained based on the actual value of the runtime. in (55), an online discriminative approach is proposed to predict runtimes for HPC applications in a parallel computing platform. In their work, they consider historical data, including few recent application runtimes as input features to online polynomial regression. They consider several settings for their model and use the available traces for manual model selection.

The Problem of Unpredictiblity

Accurate prediction of runtimes is hard. There is no guarantee that users keep the same patterns in submitting applications, and new types of application are created. Our studies show that while we can not accurately predict the application runtime, we can have a near accurate estimate of to what extent the predictions are accurate. In other words, we can predict the accuracy of the runtime prediction.

Estimating prediction reliability refers to estimating how well a prediction model will perform on unseen data. In this work, we specifically focus on estimating prediction reliability for individual out of sample data points. There have been several works in machine learning literature on determining the reliability of the prediction model for individual data points (53), (123). These approaches assume that features and target values are generated independently from the same probability distribution, and they calculate confidence for predicted target values using p-value measure. In (21), authors propose local regression sensitivity analysis to provide prediction reliability values. The application of their approaches requires multiple runs of the prediction step to determine the variance of the results and is not appropriate for our purpose of estimating accuracy values on-the-fly. Some other existing approaches are designed for a special group of prediction models and cannot be prescribed to estimate reliability for an arbitrary prediction model (110; 149; 87; 79).

Authors in (119) have reviewed prediction uncertainty for online prediction problem. They compared multiple reliability estimation using the correlation coefficient of estimated accuracy and the actual accuracy for the prediction model. In their work, the similarity-based reliability estimation approach is implemented using K-nearest-neighbors (KNN) prediction approach and is shown to be a well-performing estimation approach. Similarity-based reliability prediction approaches consider the accuracy of previous predictions of the same prediction model for similar examples in the input space. A more general approach for similarity-based prediction reliability estimation is to train a regression model that maps feature space and predicted target value to the corresponding accuracy. Using regression models to estimate prediction accuracy has been proposed in the past (52; 143; 14). Building a regression model helps to map feature set characteristics to prediction accuracy and helps to identify the characteristics of features that affect the performance of the classifier. In our work, we adopt a similarity-based approach for reliability estimation. We use gradient boosting tree regression instead of K-nearest neighbor as an accuracy prediction model and showcase its better estimation performance in comparison to the K-NN model in Section 4.

Estimating prediction accuracy has often been proposed to help model selection or iterative model improvement to achieve better prediction accuracy. However, in our work, the goal of prediction reliability estimation is whether to rely on predicted target values or not. If estimated prediction accuracy for a data point is too low, our scheduling platform ignores the predicted runtime and uses the user requested runtime with a less runtime accuracy sensitive scheduling policy.

2.3.2 Resource usage prediction for Applications and Virtual Machines

Distributed system providers need to enable proactive resource provisioning for effective resource management. This effective resource management increases their revenues. Both in private clusters and in HPC as a Service Cloud systems, customers request for specific amount of resources when they submit their applications. One important observation is the fact that customers don't use all the resources they request. Trying to predict the actual resource consumption by each virtual machine and using it as a base for proactive elastic resource provision will improve efficiency. Distributed system providers need to predict increases and decreases in resource consumption in individual VM level for efficient scheduling of workloads on resources in their clusters. Workload prediction plays a key role in proactive provisioning approaches and has been characterized as a hard problem (77).

The objective is to predict individual VM CPU consumption multiple time steps in future, given historical aggregate CPU usage. The input to the model can be expressed as

$$y = \{y_0, y_1, ..., y_{M-1}\} \tag{2.2}$$

where y_t is the aggregate CPU measurement for time step t. The predicted CPU consumptions can be expressed as:

$$\hat{y} = \{\hat{y}_M, \hat{y}_{M+1}, ..., \hat{y}_T\} \tag{2.3}$$

Two different categories approaches are used to predict time series:

1. Considering previous values as features and building input vectors from multiple recent values and applying common machine learning approaches to predict future values based on input features.

2. Applying time series specific approaches such as ARIMA that automatically consider temporal autocorrelation with previous values of the time series.

The problem with the first category of approaches is the fact they treat each of the previous values equally. For instance, if we input three recent values as input to an ANN, the model treats the immediate past value similar to the value with lag 2. This leads to loss of valuable information about correlations consecutive lags. Recurrent neural networks purposefully designed to learn all these patterns from sequences.

ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration. This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- AR (Autoregression):A model that uses the dependent relationship between an observation and some number of lagged observations.

- I (Integrated): The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- MA (Moving Average): A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

The parameters of the ARIMA model are defined as follows:

- p: The number of lag observations included in the model, also called the lag order.
- d: The number of times that the raw observations are subtracted from their lagged observations also called the degree of differencing.
- q: The size of the moving average window, also called the order of moving average.

In predictions with ARIMA, a linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model.

One limitation of ARIMA approaches is the requirement of parameter setting. We want to apply our prediction model to predict individual virtual machine traces. Using ARIMA, we need to figure out the value of p, d, and q for each VM. Additionally, As our CPU utilization trace may not be consistent throughout VM lifetime, we need to update the parameter p, d, and q for each prediction. Another issue is the fact that ARIMA builds a linear regression model on the processed lags and is not able to learn nonlinear patterns. Consequently, although ARIMA may give good predictions for next couple of time series value, it is not able to give an accurate long-term prediction.

Finding repetitive patterns and mining the correlation between the patterns have proved to be an effective approach in predicting individual VM CPU usage trace (26; 80). Finding both autocorrelation between different windows of single trace as well as correlation between windows in different traces help us to predict the values of CPU usage in future time steps. In (80), co-clustering of subsequences in traces to characterize the most repeating subsequences and then apply hidden Markov Models to characterize the temporal correlations in the the discovered clusters and use these informations to predict variations of VM workload patterns.

Different approaches are proposed to predict VM CPU usage. In (59), authors consider two different possible models to predict VM traces: signature-driven and state-driven models. They consider a cyclic pattern for a newly deployed VM and switch to state-driven model if they fail to find a signature after several resource consumption reports from the specific workload. In (105) each workload trace is decomposed into several wavelet signals and perform prediction for each signal separately. In (34), authors cluster VM trace patterns into several clusters called workload

categories and use a stochastic model to predict CPU demand for each of the workload categories. A little different from most of the previous work, (80) identify groups of VMs that show recurring patterns. They train hidden Markov Model that utilizes temporal correlations in co-cluster patterns. Inspired by their work, we propose using attention-based recurrent neural networks to find co-clusters and extract the correlation between patterns automatically. In fact, recurrent neural networks have been proposed before for the different problem of aggregate workload CPU prediction before (65; 131; 42). In (42), authors proposed the usage of recurrent neural networks for prediction of host CPU utilization. In (65), the cloud overall CPU utilization is predicted with bidirectional multivariate input LSTM networks. The problem of predicting aggregated workload is an easier problem first because only a single aggregated trace is considered instead of various individual VM traces with various patterns. Second, the aggregate trace has fewer noise (80).

Recurrent neural networks are specifically efficient for sequential data and are appropriate for cloud virtual machine consumption trace time series. More specifically, we will study Long Short-term Memory deep learning networks and show how they outperform existing approaches for predicting aggregate CPU usage by virtual machines as well as resource usage for individual workloads. CPU consumption at the level of the host has been studied extensively in the past. For this purpose, the aggregated trace of CPU usages is studied as a single time series, and future values are predicting based on the past and current values. In (59), authors use Markov Models to model resource usage on the host. In (105), authors use wavelet transformation of CPU usage trace as input to the Markov model to predict future values. (159) apply ARIMA to predict future values of for future values. (25) uses neural networks to predict future resource usage. We propose using attention-based LSTM recurrent neural networks as an end to end tool that extracts similarities between subsequences and finds correlation and auto-correlation among these subsequences by the help of LSTM gates parameters of which will be learned with back-propagation through time. We consider attention-based LSTM encoder decoders to make sure important information in hidden layers of LSTM network is not lost (8; 116). In addition to existing dual-level attention model for time series prediction, we also design a new attention model biases based on the structural consistency introduced by (80).

Chapter 3

Scheduling non-preemptive applications with varying runtimes

In this chapter, we consider the problem of scheduling jobs with non-uniform resource requirement on multiple machines. Each machine has a certain computing capacity and can schedule multiple jobs with two orthogonal dimensions of duration and resource requirement simultaneously as long as the jobs' total resource requirement does not exceed the machine's capacity. This scenario arises commonly in virtual machine scheduling in cloud computing data centers. We study this problem with the requirement that jobs must be scheduled non-preemptively, meaning every job must be completed without interruption once it gets started. We focus on the popular objective of minimizing total completion time of jobs. This problem is NP hard where the main difficulty comes from non-trivial interaction between two orthogonal quantities, jobs demands and sizes. We propose novel algorithms with provable approximation guarantees for scheduling jobs with non-uniform demands on multiple homogeneous machines without preemption. For the first time, we develop algorithms that are constant approximation for all instances. We demonstrate the superior performance of our algorithms via simulation experiments. Prior to our work, there was not any algorithm available with approximation guarantees for this problem even in the single machine case.

3.1 Introduction

Modern data centers consist of a large number of servers to handle explosive growth of data. Further, each server is getting increasingly powerful with more resources. For example, multi-processor chips have become dominant as the uniprocessor chip design has hit the thermal wall by producing too much heat to be cooled down economically. It is widely expected that more processors will be packed into a single chip in the future (56).

As servers are getting more powerful, each server is often set to process multiple jobs simultaneously to exploit its resources to the full capacity. This trend is observed

in various forms. Virtualization technologies such as VMware products and Xen allow each individual physical server/machine to be shared by multiple virtual machines. Virtualization helps reduce the cost of maintenance, operation and provision (con; 19), and hence has been adopted in current platforms including Amazon EC2 (ec2) and Microsoft Azure (azu). These platforms aim at providing ubiquitous access to shared resources where resources are shared by multiple jobs and clients. In MapReduce (36) (or its open source implementation, Hadoop) and Spark (157), which are now the de facto large data processing frameworks, typically several tasks are processed on the same server simultaneously. See (103; 126; 158) for characterizations of such tasks in Google clusters.

All the above settings can be naturally modeled as follows. Each job j has a certain computing requirement, d_j, which we call j's demand, and can be processed on a server/machine with other jobs if the total demand of the jobs does not exceed the machine's computing capacity. This model is introduced in (49). One important assumption made in (49) was that jobs can be preempted with no penalties and no delay. However, significant overhead could occur when preempting jobs to schedule other jobs. Preemption can be very costly due to context switching overheads, or may not be allowed due to system restrictions or contracts with clients. Unfortunately, non-preemption makes scheduling decisions significantly more challenging, and previous work has focused on preemptive scheduling except for some special cases. In reality, only a subset of jobs may have to be processed non-preemptively. In this chapter we focus on developing algorithms that schedule all jobs non-preemptively. A more general setting where preemption is not allowed or limited for a subset of jobs will be studied in future work. In this chapter we study *non-preemptive* scheduling algorithms in the presence of *multiple identical machines* where multiple jobs can be scheduled simultaneously subject to the capacity constraints of individual machines.

While there are various scheduling objectives considered in practice and in the literature, we consider the popular objective of minimizing the total completion time of jobs. We assume that all jobs are available for scheduling from the beginning and all jobs information is known in order to focus on the offline scheduling environment where jobs have varying demands. This problem is NP hard, and we aim at developing heuristic algorithms with *approximation guarantees*. An algorithm is said to be a c-approximation or have an approximation factor/ratio c if the algorithm's objective is at most c times the optimal scheduler's objective for *all* instances. We note that it is the non-preemptive requirement that makes the problem challenging since otherwise one can easily get a constant approximation using linear programming. To the best of our knowledge, there has been no theoretical study of this problem despite its wide appearance in practice. In this chapter, we consider the static scheduling of non-preemptive jobs on a parallel system of servers. We use the information about job sizes and resource demands to allocate jobs on servers more efficiently. Important applications include assigning virtual machines to servers.

In Section 3.2, we present the problem definition and our algorithms as well as analytical results. We elaborate on the analytic results in Section 3.3. We present

the simulation results in Section 3.4 and conclude our work in Section 3.5.

3.2 Proposed scheduling algorithms

3.2.1 Formal problem definition

There are N jobs, $1, 2, 3, ..., N$ that are to be scheduled on M servers/machines, which are indexed by $1, 2, 3, ..., M$. Each job j has processing time/size p_j and demand d_j, which are assumed to be integers. Time is slotted into unit-sized slots, $0, 1, 2, \cdots$, and jobs can start only at integer ties. It is assumed w.l.o.g. that $p_j \geq 1$. We call the quantity $v_j := p_j d_j$ as j's volume. We will be interested in non-preemptive scheduling, meaning that a job must be processed without interruption until completion once it gets started. Each job must be assigned to a server. All servers are available with full capacity of resource at time zero. A feasible schedule can be described by $\sigma = \{(S_j^\sigma, m_j^\sigma)\}_{j\in[N]}$ where S_j^σ is j's start time and m_j^σ is the server to which j is assigned. When the schedule σ is clear from the context, we may drop σ from the superscript and simply use S_j and m_j. Note that each job j's completion time $C_j = S_j + p_j$ is determined by its start time in the non-preemptive setting. For visualization of job and server, see Figure 3.2.1 where we view each job as a two dimensional rectangle with height equal to its demand d_j and width equal to its size p_j. Jobs run on the assigned server for the number of time steps equal to the job size. We say that a schedule σ is feasible if the total demand of jobs processed on each server is at most the server's capacity at all times, i.e. $\sum_{j:t\in(S_j,S_j+p_j),m_j=i} d_j \leq D_i$ for all servers i and time t where D_i denotes server i's capacity. Assuming that all servers are identical, by simple scaling, we can assume w.l.o.g. that $D_i = 1$ for all servers i and $0 < d_j \leq 1$ for all jobs j. The goal is to find a feasible schedule that minimizes total completion time of all jobs, i.e. $\sum_{j\in[N]} C_j$.

3.2.2 Our results

In this section we present our proposed algorithms as well as our main theoretical results. We start by introducing priority based algorithms for capacitated machines which are a combination of sorting jobs using different priority rules and appropriately designed machine selection schema. The algorithms are intuitive and very easy to implement. We show that one of the priority based algorithms is a constant approximation if every job has demand considerably smaller than the servers' capacities. Then, we give two different algorithms with constant approximation ratios for all inputs. The first one is a combination of two priority based algorithms. The second is based on consolidating jobs of similar size into a single job. To the best of our knowledge, these are the first algorithms with non-trivial approximation guarantees for non-preemptive scheduling of capacitated machines

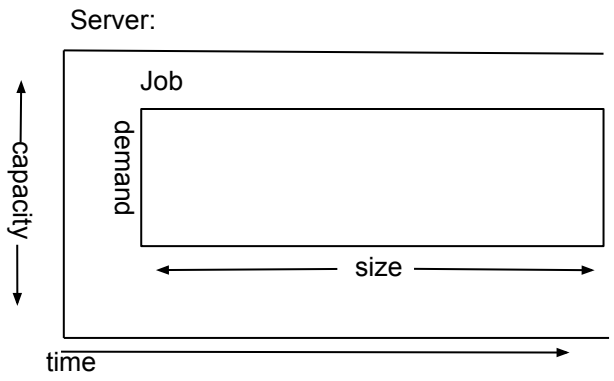

Figure 3.1: figure
Each job j is illustrated as a two dimensional rectangle with height equal to its demand d_j and width equal to its size p_j. Each server has a unit capacity. Jobs can run simultaneously on each server as long as the total demand/height of running jobs do not exceed the server's capacity.

Priority-based algorithms

Priority-based algorithms refer to those that prioritize jobs based on fixed quantities. Such algorithms are desirable in practice in that priorities remain the same among jobs and are easy to implement. In the following priority-based algorithms, jobs are ordered in non-decreasing order of the following respective quantities.

- Shortest Job First (SJF): size p_j.
- Smallest Demand First (SDF): demand d_j.
- Smallest Volume First (SVF): volume $d_j p_j$.

In the standard/incapacitated scheduling setting where a machine can schedule at most one job at a time, a complete ordering of jobs decides the entire schedule when there is a single machine, i.e, $M = 1$: the highest-priority job that is unscheduled yet is started at the earliest time when the machine becomes available. We can naturally extend priority-based algorithms to the capacitated setting where a machine can process multiple jobs simultaneously by starting the highest-priority unscheduled job j at the earliest time t when the job gets enough resources for p_j time steps from the time so that it can be processed until its completion without interruption. Each priority-based algorithm can be coupled with a machine assignment rule which decides the machine each job is scheduled on. In this work, we will consider the three machine assignment rules. Keep in mind that jobs assigned to the machine are scheduled following a fixed priority rule.

- Earliest Feasible machine (EF): Each job is assigned to the server that can start the job the earliest.

- Lowest Workload First (LW): Each job is assigned to the machine with the lowest workload.
- Random (RANDOM): Each job is assigned to a random machine.

Intuitively, EF is the most suitable rule for our goal of minimizing total completion time and also outperforms other rules in our simulation experiments. For choosing the best machine for each job j EF will check the earliest time it can be scheduled on each machine and assign it to the machine with the smallest such value. We present a psuedocode for EF algorithm in Algorithm 1 and illustrate it in Figure 3.2.

Algorithm 1 Algorithm for Earliest Feasible procedure

Job j with demand d_j and sorted list L of currently running jobs ENSURE $\sigma_j = \{S_j, m_j\}$ where S_j is j's starting time and m_j is the machine j is assigned to FOR each job k on the sorted list L STATE consider the machine m on which the job k is running on IF the current available resource on m is greater than d_j STATE assign job j to machine m STATE set job j's start time to k's completion time ($s_j \longleftarrow c_k$) STATE break for loop ENDIF ENDFOR

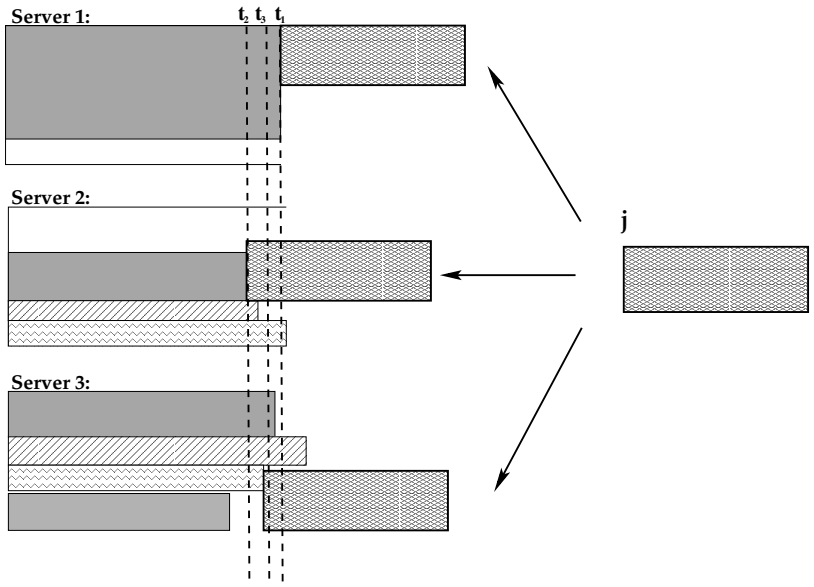

Figure 3.2: Earliest Feasible algorithm, assigns each job j on the server with the earliest feasible time. The earliest feasible time on each server is determined and the server with the minimum value is chosen. Here t_2 is the earliest feasible time and the job will be assigned to Server 2.

We will refer to each combination as the pair of the two algorithms' names. For example, SVF-EF is SVF coupled with EF. We begin by showing that simple priority-based algorithms can have poor performance for some instances.

Theorem 1. [Section 3.3.1] *For any constant $c > 1$, none of the algorithms SVF, SJF, SDF is a c-approximation for minimizing total completion time. Further, this is the case even when there is only a single machine.*

The lower bound instances that fail the priority-based algorithms are simple but give a useful insight on the problem. Let's first discuss SJF. It is folklore that SJF is optimal when all jobs have a unit demand (130). A drawback of SJF is that it could schedule slightly shorter jobs with large demands pushing back other jobs with tiny demands. The algorithm SDF can perform poorly for similar reasons. It may even schedule jobs in decreasing order of their sizes even when jobs have similar demands. Hence prioritizing jobs based on either their sizes or demands can lead to schedules of in qualities. On the other hand, it is not very obvious at first sight why SVF is bad since it looks like a natural generalization of SJF to the capacitated setting. Our lower bound instance shows that the bad event can occur when some jobs with demands close to 1 block machines for long thus delaying all jobs with very tiny demands. However, in practice not many jobs will have demands very close to machines capacities, thus SVF could perform well for most instances. We formalize this intuition in the following theorem.

Theorem 2. [Section 3.3.2] *For a constant $0 < \alpha < 1$, if all jobs have demands at most α, then SVF-EF is a $\left(\frac{3\alpha}{1-\alpha} + 3\right)$-approximation. More precisely, SVF-EF's total completion time is at most $\frac{1}{1-\alpha}$ times the optimum plus $\frac{2}{1-\alpha}$ times the total size of all jobs.*

When the system is overloaded, the contribution of jobs sizes to the total completion time objective becomes negligible. Hence, Theorem 2 implies that SVF-EF's objective becomes arbitrarily close to the optimum as α tends to 0 and the system gets more overloaded. We note that SVF was analysed only together with EF since it does not seem to yield similar approximation guarantees when coupled with other machine assignment rules.

Constant approximation algorithms

Finally, building on the intuition, along with the above results, we develop algorithms that are constant approximations for *all* instances for the first time.

Theorem 3. [Section 3.3.3] *For any constant $\epsilon > 0$, there is a $12(1+\epsilon)$-approximation when $M = 1$. When there are more than one machine, there is a $5 + O(1/M)$-approximation.*

When there are multiple machines ($M \geq 2$), we combine SJF and SVF. Let us call jobs with demands higher than $1/2$ as high-demand jobs, and others as low-demand jobs. It is easy to see that no two high-demand jobs can be processed on the same machine at the same time. Intuitively, scheduling high-demand jobs in the capacitated setting is similar to scheduling jobs in the incapacitated setting. Hence we dedicate

some machines to processing high-demand jobs separately, and process low-demand jobs on the remaining machines. Thus, by separating high-demand jobs from low-demand jobs, we achieve a constant approximation. However, when there is only one machine ($M = 1$), we need a different idea. In this case, we reduce the problem to the geometric packing problem where the goal is to maximize the total profit of rectangles that can be packed into a target rectangle (75). Using this reduction and borrowing ideas from (136), for any time t, we can obtain a partial schedule that completes as many jobs by time $3(1+\epsilon)t$ as the optimal scheduler does by time t. Then, we derive a complete schedule by concatenating the partial schedules for geometrically increasing t. We note that it is possible to reduce this problem to the maximum throughput scheduling problems (10; 6). However, the resulting approximation ratio seems worse, hence we use the above reduction.

As discussed earlier, any reasonably good algorithms seem to have to take into account both quantities, jobs sizes and demands. Then, a natural question is if we really need a delicate 2D-packing to obtain constant approximations. Quite surprisingly, we show that this is not the case by developing a constant approximation algorithm with a nearly linear running time. While the approximation guarantee is worse than the previous algorithm, we include this result since it shows the possibility of existence of more efficient algorithms with approximation guarantees comparable to those claimed in Theorem 3.

The Block-Scheduling algorithm takes advantage of the fact that rounding the job sizes to the next power of two, we do not lose more than a factor of two in the approximation ratio for the total completion time objective. The first step in the Block-Scheduling algorithm is to group jobs of similar sizes into the same class. The class k inludes jobs of size 2^{k-1} to 2^k. Then we pack jobs in each class into blocks of capacity equal to that of the servers. In order to complete as many jobs as possible sooner, we order jobs of each class in non-decreasing order of their demands before fitting them into blocks. After forming the blocks, we order them in non-increasing order of their densities. We define density of a block as the number of jobs in the block divided by its size. A block's size is equal to that of any job in the block, which is well defined as all jobs in the same block have an equal (rounded) size.

At a high-level, the Block-Scheduling algorithm is based on reduction to bin packing. The key idea is to group jobs of similar sizes so that each 'consolidated' block has a sufficiently large demand. Then, we schedule the consolidated blocks mimicking the algorithm Highest Density First (HDF). The algorithm HDF is a generalization of SJF to the case where jobs have weights, and is known to be constant-approximate in the incapacitated setting. The actual algorithm needs minor modifications, but this is a high-level description of our algorithm and the intuition. While the high-level idea is intuitive, the analysis is non-trivial.

The formal description of the Block-Scheduling algorithm is as follows.

1. Packing jobs into blocks: For each class k, create blocks using the First-Fit algorithm: Consider jobs in class k in non-decreasing order of their demands, and partition them into groups so that the total demand of jobs in every group

does not exceed 1. Here, we create another group only when the currently considered job cannot fit with other jobs into the existing groups. We call the created group as blocks, and let $B_{k,l}$ denote the lth block we created for jobs in class k.

2. Ordering blocks: Let p_B denote the size of block B, which is defined as the size of any job packed into the block. Let N_B as the number of jobs in B. Define B's density, $\eta_B := N_B/p_B$. Blocks are ordered in non-increasing order of their densities.

3. Assigning blocks to machines: We will recursively schedule blocks as follows. In the kth step, for each machine m from 1 to M, we find a maximal set $\mathcal{B}_k^m$ of unscheduled blocks of sizes at most 2^k with the highest densities such that the total size of blocks in $\mathcal{B}_k^m$ does not exceed $\gamma \cdot 2^k$, and schedule all jobs in the set $\mathcal{B}_k^m$ on the machine m, where $\gamma := (9 + \lceil 6M \rceil)$. Note that jobs are scheduled by blocks in that no two jobs from different blocks are processed at the same time on the same machine. Blocks in $\mathcal{B}_k^{A,m}$ are scheduled in non-increasing order of their densities.

Theorem 4. [Section 3.3.4] *There is a constant approximation where all jobs processed at the same time have an equal size within a factor of 2. Further, the algorithm runs in $O(N \log N)$ time.*

3.2.3 Related work

When each job has a release time and deadline and the goal is to maximize the total weight of jobs completed before their deadlines, several approximation algorithms are known (10; 6). If the objective is minimizing makespan, i.e. the maximum completion time, we can easily adapt our algorithms and analysis to obtain constant approximations using the proof ides used in Section 3.3. As mentioned before, it is well known that Shortest Job First (SJF) is optimal in the standard non-capaciated setting. However, when jobs have release dates, the problem becomes NP-hard but admits a $(1 + \epsilon)$-approximation for any $\epsilon > 0$ (28). For other results on variants of completion time objectives, see a nice survey by Chekuri and Khanna (29).

As mentioned, (49) studied our problem in the preemptive setting, and gave and analyzed several online algorithms for the total flow time objective under the resource augmentation model; a job j's flow time is defined as the job's completion time minus its arrival time. In contrast, our work studies non-preemptive scheduling. In the non-preemptive setting, the flow time objective becomes very difficult to approximate (9). Hence, we consider the completion time objective which admits more positive results. For works in the queueing setting where jobs arrive following certain distributions, see (98; 99). In such models, typically the focus is on the stability of the system, which is very different from our work.

A lot of work has been done on somewhat related problem of scheduling jobs on parallel machines. For example, see (46; 18; 121; 27; 161). Roughly speaking, in

these works, one can schedule a job on multiple machines/processors simultaneously to speed up the processing. While these models accurately capture how jobs are parallelized at a high level, they do not enforce hard capacity constraints on jobs demands. On the other hand, in our model, jobs cannot be processed when given less resources than their demands.

The capacitated machine model is close to batch scheduling of jobs in operation research. A batch processing machine model is encountered in many different environments, such as heat treatment operations in the metalworking industries and semiconductor manufacturing. The batch processing machine in that literature is one which can process a number of jobs simultaniously as a batch. All jobs in a batch start running at the same time and the completion time for each job is equal to that of the longest job in the batch. However these two models are different as in the former new jobs can start running as soon as there is enough available resource for them. For a list of works on the operation research problems see (144), (41) and (84).

On the other hand, our work is related to virtual machine scheduling. In most of the current resource management systems, cloud provider allocate virtual machines on physical servers. Bin-packing algorithms have been proposed to allocate virtual machines on servers (12), (23),(95). In these problems, server resources such as CPU, memory and bandwidth and storage are dimensions of the bin packing problems. These algorithms are often used in server selection phase to achieve cost minimization, energy efficiency and utility maximization. For the objective of minimizing the number of servers, (147) and (23) proposed approximation algorithms assuming homogeneous servers. In (95), Lin et. al applied a multi-dimensional vector packing algorithm to allocate virtual machines in a cloud system. In some works including (132), virtual migration facilitated adaptive approximation algorithms. However in all these bin packing algorithms, only the objectives of cloud provider are taken into account and customer SLA satisfaction regarding responsiveness is not taken into consideration.

3.3 Analysis

3.3.1 Lower bounds for priority based algorithms

This section is devoted to proving Theorem 1. We assume throughout this section that there is only a single machine. We note that the lower bounds for SJF and SDF hold even if jobs have demands less than any fixed constant. The lower bound instances are summarized in Table 3.1. Each instance consists of two types of jobs, A and B.

Lower Bound for SJF: Consider the following instance. There are two types of jobs. Type-A jobs have demands 1/2 and sizes 1. Type-B jobs have demands $\frac{1}{2N}$ and sizes $1+\epsilon$ where $\epsilon > 0$ is an arbitrarily small parameter. There are $\sqrt{N}$ type-A jobs, and the other $N-\sqrt{N}$ jobs are type-B. The algorithm SJF will first complete Type-A jobs by time $\sqrt{N}/2$. Since no type-B jobs get processed before time $\sqrt{N}/2$, the total

Table 3.1: Lower bounds for priority based algorithms on single server

		SJF	SDF	SVF	
type A:	sample size	$\sqrt{N}$	1	1	
	demand	1/2	$1-\epsilon$	$\frac{1}{2N^2}$	
	size	1	N	N	
type B:	sample size	$N-\sqrt{N}$	$N-1$	$N-1$	
	demand	$\frac{1}{2N}$	1	1	
	size	$1+\epsilon$	$\frac{1}{N}$	$\frac{1}{N}$	

completion time by SJF is at least $(N-\sqrt{N})\cdot\sqrt{N}/2 = \Omega(N^{1.5})$. On the other hand, we can complete all Type-B jobs by time 1, and all Type-A jobs by time $(\sqrt{N})/2+1$. Hence the optimal total completion time is at most $(N-\sqrt{N})\cdot 1+\sqrt{N}\cdot((\sqrt{N})/2+1) = O(N)$, which gives a gap of $\Omega(\sqrt{N})$.

Lower Bound for SDF: The lower bound instance is as follows. As before there are two types of jobs. The unique Type-A job has demand $(1-\epsilon)$ and size N where $\epsilon > 0$ is a parameter that is arbitrarily small. Type-B jobs have demands 1 and size $1/N$. All the N jobs are type-B except the unique Type-A job. The algorithm SDF processes no type-B jobs until time N, hence has total completion time at least $N(N-1)$. However, we can complete all type-B jobs by time 1 and the type-A job by time N+1, hence the optimal total completion time is at most $(N-1)\cdot 1 + 1\cdot(N+1) = 2N$. Hence we obtain a gap of $\Omega(N)$.

Lower Bound for SVF: Again, there are two types of jobs. The unique Type-A jobs has size N and demand $\frac{1}{2N^2}$. All the other $N-1$ jobs are Type-B and they have size $1/N$ and demand 1. The algorithm SVF starts processing the type-A jobs. Note that type-B jobs cannot be processed until the job-A completes since they require the full capacity to get processed. Hence SVF has total completion time at least $N(N-1)$. In contrast, the optimal total completion time is at most $2N$ since we can complete type-B jobs by time 1 and the type-B job by time $N+1$, which implies a gap of $\Omega(N)$.

3.3.2 Upper bounds for priority based algorithms

In this section we prove Theorem 2. Since SVF will be paired only with EF, we may refer to SVF-EF simply as SVF. Similarly, another algorithm SJF which is considered for the sake of analsis, will be paired with EF, and we will refer to SJF-EF simply as SJF. As mentioned before SJF is known to be optimal in the incapacitated case. We first show that the optimal schedule can only be better off if it can compress jobs: replace each job j with a job j' with demand 1 and size $p_j d_j$, preserving the volume of the job. Let I be the original instance and I' the compressed instance. For notational convenience, for an instance I'', we allow $\mathrm{OPT}(I'')$ to denote a fixed

optimal schedule for instance I'' or its objective depending on the context. The resulting problem will be a incapacitated scheduling for which we already know that SJF is the optimal algorithm in terms of total completion time. We first show that $\mathrm{OPT}(I')$ lower bounds $\mathrm{OPT}(I)$. We first show that the optimal schedule can only be better off if it can compress jobs.

Lemma 1. $\mathrm{OPT}(I') \leq \mathrm{OPT}(I)$.

Proof. Consider a fixed machine m. Let $j_1, j_2, ..., j_k$ be the jobs assigned to the machine m, which are ordered in their completion times in schedule $\mathrm{OPT}(I)$. Let $j'_1, j'_2, ..., j'_k$ be the compressed jobs corresponding to $j_1, j_2, ..., j_k$. We process jobs $j'_1, j'_2, ..., j'_k$ in this order on the same machine. It is easy to see that we can complete each job j'_κ before j_κ completes in $\mathrm{OPT}(I)$. This is because no schedule can complete all jobs $j_1, j_2, ..., j_\kappa$ before time $\sum_{h=1}^{\kappa} p_h d_h$ which is the total volume of jobs $j_1, ..., j_\kappa$, and compression preserves each job's volume. This completes the proof. □

Further, we know that SJF is optimal for the instance I' since I' is an instance in the incapacitated setting (115). Hence we can assume w.l.o.g. that the optimal schedule $\mathrm{OPT}(I')$ for I' is generated by SJF. Let S_j and C_j denote job j's start and completion times in the former schedule, respectively. Define S^*_j and C^*_j analogously for the latter schedule $\mathrm{OPT}(I')$. The reader may wonder if compression gives too much power to the optimal scheduler. For example, it may finish a long job with a tiny demand very quickly by compressing it. Then, $C^*_j - S^*_j$ could be much smaller than p_j. Hence we will bound C_j by S^*_j and p_j in Theorem 2.

Lemma 2. *Suppose SVF starts processing job j at time S_j. Then it must be the case that each machine's capacity is used by at least $1 - \alpha$ from time 0 to S_j.*

Proof. For the sake of contradiction, suppose that a machine is utilized up to capacity $1 - \alpha$ during the time interval (t_1, t_2) and a job start on the machine at time t_2. This is a contradiction since SVF could have started j earlier, no later than time t_1. □

Lemma 3. *For all jobs j, we have $S_j \leq \frac{1}{(1-\alpha)M} \sum_{h=1}^{j-1} v_h$.*

Proof. From Lemma 2, we know that SVF processed at least $(1 - \alpha)$ volume of work for jobs $1, 2, ..., j - 1$ by time S_j. Hence, we have $\sum_{h=1}^{j-1} v_h \geq (1 - \alpha)MS_j$. □

Lemma 4. *For all jobs j, we have $S^*_j \geq \frac{1}{M} \sum_{h=1}^{j-1-(M-1)} v_h$.*

Proof. In SJF's schedule, at most $M - 1$ jobs are being processed when job j starts getting processed. Hence SJF must complete at least $j - 1 - (M - 1)$ jobs out of $1, 2, ..., j - 1$ by time S^*_j. Thus, SJF processes at least $\sum_{h=1}^{j-1-(M-1)} v_h$ volume of work by time S^*_j, which yields the lemma. □

We are now ready to complete the proof of Theorem 2. For each job j we derive,

$$\begin{aligned}
C_j &\le p_j + \frac{1}{(1-\alpha)M}\sum_{h=1}^{j-1} v_h \quad \text{[By Lemma 3]} \\
&= p_j + \frac{1}{(1-\alpha)M}\Bigg(\sum_{h=1}^{j-1-(M-1)} v_h + \sum_{h=j-M+1}^{j-1} v_h\Bigg) \\
&\le p_j + \frac{1}{(1-\alpha)M}\Bigg(\sum_{h=1}^{j-1-(M-1)} v_h + \sum_{h=j-M+1}^{j-1} v_j\Bigg) \\
&\le p_j + \frac{1}{1-\alpha}\cdot S_j^* + \frac{M-1}{M(1-\alpha)} v_j \quad \text{[By Lemma 4]} \\
&\le \frac{1}{1-\alpha}\cdot S_j^* + \frac{2}{1-\alpha}\cdot p_j
\end{aligned}$$

The last inequality follows since job j has demand at most α, meaning $v_j \le \alpha p_j$. Knowing that $\sum_j S_j^*$ and $\sum_j p_j$ are at most the optimal total completion time, we have Theorem 2.

3.3.3 Constant approximation algorithms

In the previous section we showed that SVF is a constant approximation if all jobs have demands at most α that is a constant smaller than 1. In this section, we develop algorithms that are $O(1)$-approximation for all instances, proving Theorem 3. We consider two cases depending on whether $M \ge 2$ or not. **Single Machine Case ($M = 1$):** In this case, we reduce our problem to the 2D-strip packing problem. Using this reduction, we show the following.

Lemma 5. *Suppose there is a subset of n jobs that can be completed within L time steps. Then, for any constant $\epsilon > 0$, one can find a schedule in polynomial time that completes at least n jobs by time $3(1+\epsilon)L$.*

We can complete as many jobs as the optimal scheduler if we are allowed to use $3(1+\epsilon)$ factor more time steps. Before proving the lemma, we first discuss how we use it to prove Theorem 3 in the single machine case. By repeatedly using this lemma, we obtain partial schedules and concatenate them to get a final schedule. Let N_ℓ denote the total number of jobs that the optimal schedule completes by time 2^ℓ. Using Lemma 5, we find a set of at least N_ℓ jobs that can be scheduled by time $3(1+\epsilon)2^\ell$. Let this schedule be denoted by B_ℓ, which we call a block. We concatenate the blocks $B_0, B_1, B_2, ...$ in this order. If a job in B_ℓ was already scheduled in the previous block, we simply remove the job from the block B_ℓ. The resulting schedule is clearly feasible. Note that jobs in block B_ℓ are processed between times $3(1+\epsilon)\sum_{h=0}^{\ell-1} 2^h$ and $3(1+\epsilon)\sum_{h=0}^{\ell} 2^h = 3(1+\epsilon)\cdot(2^{\ell+1}-1)$. For notational simplicity, let $N_{-1} := 0$. In this schedule, the total completion time is at most

$$3(1+\epsilon)\sum_{\ell\ge 0} 2^\ell (N - N_{\ell-1}) \tag{3.1}$$

This is because when the block B_ℓ is scheduled, there are at most $N - N_{\ell-1}$ jobs alive, and the block is scheduled for $3(1+\epsilon)2^\ell$ time steps. On the other hand, the optimal total completion time is at least

$$N + \sum_{\ell \geq 1} 2^{\ell-1}(N - N_\ell), \tag{3.2}$$

since the optimal schedule has at least $N - N_\ell$ jobs alive during the time interval $(2^{\ell-1}, 2^\ell)$ for $\ell \geq 1$. The first term N follows since no job completes before time 1; recall that all jobs have sizes at least 1. A simple algebra gives Theorem 3 for the case $M = 1$. It now remains to prove Lemma 5.

: We borrow ideas from (75) which studies the problem of maximizing the total profit of rectangles that can be packed into a target rectangle without overlap; here rectangles can only be moved vertically and horizontally, and rotations are not allowed. We first find a set of jobs J whose total volume does not exceed $(1+\epsilon)L$. This is essentially a special case of the Knapsack problem where we are asked to pack items of different profits and sizes into a knapsack with the goal of maximizing the total profit of items packed. Then, it is well known that we can pack in polynomial time as many items into a knapsack of size $(1+\epsilon)L$ as the optimal solution can pack into a knapsack of size L; for example see (150). We now view the scheduling instance as an input to the 2D-strip packing problem. In the 2D-strip packing problem, we are asked to pack all given rectangles without rotations into a strip with bounded width but with unbounded height, and the goal is to minimize the strip height. Towards this end, for each job j, create a rectangle $r(j)$ with width p_j and height d_j. Steinberg (136) shows a sufficient condition that all rectangles can be packed – particularly, the condition is satisfied if the strip height is at least twice the maximum height of rectangles, and the total area of rectangles is at most half of the strip area (and with other 'easy conditions'). Hence we can pack all rectangles corresponding to J into a strip with width $(1+\epsilon)L$ and height 2. Following ideas in Section 3 of (75), we can pack all the rectangles corresponding to J into three strips with width $(1+\epsilon)L$ and height 1. We obtain the desired schedule by concatenating these three strips horizontally and translating it into a schedule. **Multiple Machine Case ($M \geq 2$):** We schedule jobs of demands more than 1/2 (high-demand jobs) and the other jobs (low-demand jobs) separately on two disjoint sets of machines, $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. Our new algorithm, which we call HYBRID, combines two algorithms, SJF and SVF. We use SJF to schedule high-demand jobs on $\mathcal{M}_1$ pretending that high-demand jobs have demands equal to 1. For low-demand jobs, we schedule them on $\mathcal{M}_2$ using SVF. As in the proof of Lemma 1, we use S_j and C_j to denote job j's starting and completions times in the schedule of our algorithm.

Low-demand Jobs: We first upper bound low-demand jobs' contribution to the objective. Obviously, the optimal scheduler can only decrease its objective if it only needs to complete low-demand jobs. Hence we can assume w.l.o.g. that we only have low-demand jobs. The analysis is similar to that in the previous section. The only

difference is that SVF can only use M_2 machines while the optimal scheduler can use all M machines. As before, let S_j^* and C_j^* denote the start and completion times of a low-demand job j in the optimal schedule respectively, assuming that all jobs are compressed. Then, we can easily adapt Lemmas 3 and 4 as follows.

Lemma 6. *For all low-demand jobs j, we have*

- $S_j \leq \frac{2}{M_2} \sum_{h=1}^{j-1} v_h$.
- *For all jobs j, we have* $S_j^* \geq \frac{1}{M} \sum_{h=1}^{j-1-(M-1)} v_h$.

Lemma 7. *For all low demand jobs j, we have* $C_j \leq \frac{2M}{M_2} \cdot S_j^* + \left(\frac{M-1}{M_2} + 1\right) \cdot p_j$

Proof.

$$\begin{aligned}
C_j &\leq p_j + \frac{2}{M_2} \sum_{h=1}^{j-1} v_h \quad \text{[Lemma 6]} \\
&= p_j + \frac{2}{M_2} \left(\sum_{h=1}^{j-1-(m-1)} v_h + \sum_{h=j-m+1}^{j-1} v_h \right) \\
&\leq p_j + \frac{2}{M_2} \left(\sum_{h=1}^{j-1-(m-1)} v_h + \sum_{h=j-m+1}^{j-1} v_j \right) \\
&\leq p_j + \frac{2M}{M_2} \cdot S_j^* + \frac{2(M-1)}{M_2} v_j \quad \text{[Lemma 6]} \\
&\leq p_j + \frac{2M}{M_2} \cdot S_j^* + \frac{M-1}{M_2} p_j \quad \text{[Since } v_j \leq \tfrac{1}{2} p_j \text{]} \\
&\leq \frac{2M}{M_2} \cdot S_j^* + \left(\frac{M-1}{M_2} + 1 \right) \cdot p_j
\end{aligned}$$

□

High-demand Jobs: We can assume w.l.o.g. that the optimal scheduler only needs to complete high-demand jobs on the M machines. For notational convenience, assume that high-demand jobs $1, 2, 3, ...$ are ordered in non-decreasing order of their sizes. Knowing that at most one high-demand job can be processed at any time, we can also assume w.l.o.g. that optimal schedule is produced by SJF.

Lemma 8. *For all high-demand jobs j, we have*

- $S_j \leq \frac{1}{M_1} \sum_{h=1}^{j-1} p_h$.
- *For all jobs j, we have* $S_j^* \geq \frac{1}{M} \sum_{h=1}^{j-1-(M-1)} p_h$.

Proof. The upper bound on S_j follows from the fact that at most one machine gets available for scheduling job j before time $\frac{1}{M_1} \sum_{h=1}^{j-1} p_h$. The lower bound on S_j^* follows since SJF completes all jobs shorter than j but possibly other jobs being processed on the other $M-1$ machines at time S_j^*. □

Lemma 9. *For all high demand jobs j, $C_j \leq (\frac{M-1}{M_1} + 1) \cdot C_j^*$*

Proof.

$$\begin{aligned} C_j &\leq p_j + \frac{1}{M_1} \sum_{h=1}^{j-1} p_h \quad \text{[Lemma 8]} \\ &= p_j + \frac{1}{M_1} \left(\sum_{h=1}^{j-1-(M-1)} p_h + \sum_{h=j-M+1}^{j-1} p_h \right) \\ &\leq p_j + \frac{M}{M_1} \cdot S_j^* + \frac{M-1}{M_1} p_j \quad \text{[Lemma 8]} \\ &\leq (\frac{M-1}{M_1} + 1) \cdot C_j^* \end{aligned}$$

In the last inequality we used the fact that $C_j^* = S_j^* + p_j$. □

Putting the Pieces Together: By summing the inequalities in Lemma 7 over all low-demand jobs, we have that HYBRID is a $(\frac{3M-1}{M_2} + 1)$-approximation for low-demand jobs. By summing over the inequalities in Lemma 9 over all high-demand jobs, we have that HYBRID is a $(\frac{M-1}{M_1} + 1)$-approximation for high-demand jobs. We set $M_1 = \lfloor \frac{M-2}{4} \rfloor + 1$ and $M_2 = \lfloor \frac{3(M-2)}{4} \rfloor + 1$. A simple algebra gives Theorem 3 for the multiple machine case.

3.3.4 Upper bounds for Block-Scheduling algorithm

The constant approximations we gave in the previous section, try to pack jobs efficiently into machines. In particular, in the single machine case, the algorithm uses a geometric packing. In this section, we show that even if we only process jobs of similar size simultaneously, we can obtain constant approximations. Before we describe our algorithm, we do the following simple preprocessing. The loss in the approximation ratio will be factored in at the end of analysis.

Proposition 1. *We can assume w.l.o.g. that each job size is a power of two with a loss of factor two in the approximation ratio.*

Proof. For each job j with size $p_j \in (2^{k-1}, 2^k)$, we pretend that the job has a size of 2^k for our algorithm and 2^{k-1} for the optimal scheduler; let's call the modified instances the algorithm and the optimal scheduler have to handle I_A and I_O, respectively. This assumption is w.l.o.g. since this can only hurt our algorithm while benefiting the optimal scheduler. Further, we can easily transform our algorithm's schedule for the I_A into a schedule for the original instance without increasing the objective, by keeping each job's start time the same. Note that the only difference between I_A and I_O is that each job's size in I_A is twice that in I_O. This implies a one-to-one mapping between schedules for the two instances I_A and I_O. More precisely, if we stretch a schedule for I_O over time by a factor of two, we obtain a schedule for I_A exactly doubling the objective (a job is scheduled at time t in the former schedule if

and only if it is scheduled at time $2t$ in the latter schedule.) Thus, we can assume w.l.o.g. that our algorithm and the optimal scheduler are asked to schedule the same set of jobs losing the approximation ratio by a factor of two. □

For the sake of analysis, we assume the optimal scheduler is allowed to compress jobs but in a somewhat restrictive manner. Recall that for each job j, its compressed version has demand 1 and size $p_j d_j$. The restriction we impose here on the optimal scheduler is that it can not complete any job of size greater than t by time t. This is w.l.o.g. because it only strengthens the optimal scheduler. For the rest of the proofs, we consider this strengthened optimal scheduler and call it OPT.

We use the well-known fact that the total completion time of jobs is equal to the sum of the number of alive jobs over all time steps. This equivalent view immediately follows by observing each alive/unfinished job contributes to the objective by one at each time step. Let's take a close look at the optimal scheduler using this lens. The optimal scheduler's objective can be expressed as $\sum_{t\geq 0} R_t^*$ where R_t^* denote the number of jobs alive at time t in the optimal scheduler. Thus, we can view R_t^* as the penalty that the optimal pays at time t. For *each* time t, we let the optimal scheduler minimize the number of jobs alive at time t. That is, the OPT's only goal is to try to complete as many jobs as possible by time t without taking the global schedule into account. This only gives additional power to the optimal scheduler. Similarly, We let R_t denote the number of jobs alive at time t in the schedule of our algorithm, which we refer to as A. We will show the following key lemma, which immediately imply Theorem 4.

Lemma 10. *For all times $t \geq 0$, $R_{2\gamma t} \leq R_t^*$.*

[Theorem 4]

$$\int_{t=0}^{\infty} R_t \; dt = 2\gamma \int_{t=0}^{\infty} R_{2\gamma t} \; dt \leq \int_{t=0}^{\infty} R_t^* \; dt.$$

Knowing that the first [last, resp.] integral is A's [OPT's, resp.] total completion time, and factoring in the approximation loss stated in Proposition 1, we derive that A is a 4γ-approximation where $\gamma := (9 + \lceil 6M \rceil)$. It now remains to prove Lemma 10.

Consider a fixed time t and let $\mathcal{B}^*$ be the set of blocks containing all jobs $j \in G_k$ that OPT completes by t. Note that for each k, there exists l_k^* such that OPT completes all jobs in $B_{k,1}, B_{k,2}, ..., B_{k,l_k^*-1}$ but no job in $B_{k,l_k^*+1}, B_{k,l_k^*+2}, ...$ We assume w.l.o.g. that OPT completes all jobs in B_{k,l_k^*} with no cost added to objective function; this is w.l.o.g. since it only gives more power to OPT. We call such blocks B_{k,l_k^*} bonus blocks. If A completes all blocks in $\mathcal{B}^*$ by time $2\gamma t$, the lemma trivially holds true for the time t, so suppose not. Let $\mathcal{B}$ be the set of blocks A completes by time $2\gamma t$. Let $\eta^* := \max_{B \in \mathcal{B}^* \setminus \mathcal{B}} \eta_B$. Let k^* be the largest k such that $2^k \leq t$; so we have $2^k \leq t < 2^{k+1}$. Note that A completes the k^*th step by time $2\gamma t$ on every machine. We assume that there is at least one block of size at most 2^{k^*} that A couldn't schedule by the k^*th step since otherwise Lemma 10 trivially holds. We will distinguish blocks

into two types. Let J_B denote the jobs in block B. We say that a block B is *half-full* if the total demand of jobs in J_B is at least half, otherwise *almost-empty*. The following two propositions are immediate from the fact that blocks are constructed by the First-Fit algorithm.

Proposition 2. *For each half-full block B, it is the case that $\sum_{j \in J_B} p_j d_j \geq p_B/2$.*

Proposition 3. *No two almost-empty blocks have an equal size.*

We will prove the following two lemmas.

Lemma 11. *The total number of jobs that A completes on all machines by time $2\gamma t$ other than jobs in $\mathcal{B} \cap \mathcal{B}^*$ is at least $\eta^* \cdot (4M+2) \cdot 2^{k^*}$.*

Proof. The total size of almost-empty blocks in $\mathcal{B}^*$ that are scheduled in the k^*th step is at most $2^0 + 2^1 + 2^2 + ... + 2^{k^*} \leq 2 \cdot 2^{k^*}$, since all scheduled blocks in the k^*th step have sizes at most 2^{k^*} and due to Proposition 3. Also we know that the total size of half-full blocks in $\mathcal{B}^*$ is at most

$$2tM + (2^0 + 2^1 + 2^2 + ... + 2^{k^*}) \leq (4M+2) \cdot 2^{k^*}$$

To see this, note that the total volume of jobs in half-full blocks in $\mathcal{B}^* \setminus \bigcup_k B_{k,l_k^*}$ is upper-bounded by t on each machine, and by Proposition 2, the total size of those blocks is at most $2t$. Also recall that we added at most one bonus block of size 2^k to $\mathcal{B}^*$ for each k, which is B_{k,l_k^*}. We also know that the algorithm schedules blocks of total size at least $(\gamma - 1)2^{k^*}$; the loss '-1' is upper bounded due to the fact that only blocks of sizes at most 2^{k^*} are considered in the k^*th step. All of these mean that the total size of blocks in $\mathcal{B} \setminus \mathcal{B}^*$ in the k^*th step is at least

$$M(\gamma - 1) \cdot 2^{k^*} - 2 \cdot 2^{k^*} - (4M+2) \cdot 2^{k^*} = (4M+2) \cdot 2^{k^*}$$

We also know that for every block $B \in \mathcal{B} \setminus \mathcal{B}^*$, $\eta_B \geq \eta^*$ since A chose to schedule B over the block B' in $\mathcal{B}^* \setminus \mathcal{B}$ with $\eta_{B'} = \eta^*$. This means that the number of jobs that A finished in the k^*th step but OPT didn't is at least $\eta^* \cdot (4M+2) \cdot 2^{k^*}$. □

Lemma 12. *The total number of jobs that* OPT *completes by time t on each machine other than jobs in $\mathcal{B} \cap \mathcal{B}^*$ is at most $\eta^* \cdot (4M+2) \cdot 2^{k^*}$.*

Proof. Consider each half-full block $B \in \mathcal{B}^* \setminus \mathcal{B}$. We know that the total number of jobs in B is at most $\eta^* p_B$. Also by Proposition 2, we know that B's volume (the total volume of jobs in B) is at least $p_B/2$. What this implies is that each unit time processing can (fractionally) complete at most $2\eta^* M$ jobs in half-full blocks in $\mathcal{B}^*$. So, for fixed time t, the total number of jobs in half-full blocks $\mathcal{B}^*$ is at most $2\eta^* Mt \leq \eta^* \cdot 4M \cdot 2^{k^*}$. Since there is at most one almost-empty block in $\mathcal{B}^* \setminus \mathcal{B}$ of size 2^k for each k, and its density is at most η^*, it follows that the total number of jobs in almost-empty blocks in $\mathcal{B}^* \setminus \mathcal{B}$ is at most $\eta^* \cdot 2 \cdot 2^{k^*}$. □

The above two lemmas imply that A complete as many jobs by time $2\gamma t$ as OPT does by time t, which proves Lemma 10.

3.4 Simulation experiments

In (Im et al.), we conducted a performance analysis of different priority based algorithms and SVF-EF proved to have the best performance. In this section, in addition to comparing the performance of different priority based algorithms, we perform experiments on HYBRID and Block-Scheduling algorithms and compare their performance with the best performing priority based algorithm, SVF-EF. In the new experiments, we study the trend in the total completion time as the numbers of jobs and servers increase. The simulation methodology is similar to the (Im et al.). We emphasize that although our workload simulator is generating workloads with specific distributions, our algorithms are proved in Section 3.3.3 to do well for any arbitrary workload. However, the choice of input workload is for the purpose of better simulation of the real world systems.In all experiments we consider uniform machines each with capacity of 200 CPU resources. We developed a discrete-event simulator in Java which consists of parallel machines with bounded capacities. Each job is described by job size and a number for CPU requirement. We assume that preemption is not allowed and all jobs are available at the beginning.

3.4.1 Workloads

Similar to (Im et al.) and following the same convention in (99) and (98), we assume jobs' CPU demands are distributed uniformly in interval $[1, 100)$ with probability 0.75 and in interval $[100, 200]$ with probability 0.25. For job sizes we assumed two distributions:

1. SYNTH1: Distributed uniformly in interval $[1, 100]$ with probability 0.7, in interval $[300, 350]$ with probability 0.15, and in interval $[450, 500]$ with probability 0.15 .

2. SYNTH2: Distributed geometrical with mean equal to that of the uniform distribution for SYNTH1.

We performed all experiments using both SYNTH1 and SYNTH2 datasets. As the results of the two distribution followed the same pattern, we are only including SYNTH2 results for better readability of some of our plots and tables (Table 3.2, Figure 3.6, Figure 3.4.2).

We also performed experiments with a real world workload dataset from an online benchmark repository (wor). We chose the HPC2N dataset which includes job sizes and CPU requirements. We ignored details like job submission times included in SWF format (48) as we assume all jobs are available for schedule at the beginning. We run our algorithms on a sample of 800 jobs randomly chosen from HPC2N.

3.4.2 Experimental results

In this section we first evaluate the performance of priority based algorithms using synthetic and the real world data sets. Then we will show case the HYBRID and Block-Scheduling algorithms by comparing them to the best priority based algorithm.

In the first experiment, we compare the performance of the algorithms as a function of workload size, i.e., the number of jobs. We generated Workload sizes from 3000 to 16000 in an increment of 1000 for both SYNTH1 and SYNTH2. For each workload size we generated data 10 times and repeated the experiments for each dataset. Figure 3.3 shows average completion time for jobs vs. number of jobs using the synthetic workload. This experiment was conducted by running synthetic workloads on 50 machines. The relative performance of different priority rules are similar in both distributions. However, in plots for SYNTH2, SVF-EF diverges from other adjacent curves more rapidly as can be seen in Figure 3.3.b. Compatible with our theoretical findings (Theorems 1 and 3), SVF-EF demonstrates the best performance, considerably outperforming SJF-EF. Further, SVF-EF's superiority in performance becomes more prominent as the workload size increases. As some plots were overlapping, we zoomed into that portion of the graph for better illustration. In second experiment we tested our algorithms on a sample of 800 jobs randomly selected from a real-world trace HPC2N. The comparison of average completion times is shown in Figure 3.5. It can be seen that SVF-EF has the best performance for the real-world data sets.

Fairness is another important scheduling criterion in high performance computing systems. As we are minimizing average completion time, it could happen that larger jobs get more benefits than smaller jobs as they contribute more to the objective function. A common measure for fairness in the literature is *stretch* which measures by what factor, a job is slowed down relative to the time it takes to complete on an unloaded system (137; 13; 135).

In another experiment, we compared the average, maximum and standard deviation of the different algorithms for 5000 jobs generated from SYNTH1 on 10 machines and the results are shown in Table 3.2. It can be seen that SJF-EF and SVF-EF have the best average and maximum stretches. The SJF-EF performs slightly better than SVF-EF, as sorting jobs in their order of increasing sizes counteracts the effect of discriminating against small jobs. However, SVF-EF does quite a good job in terms of fairness considering large gap between the performance of SVF-EF and SJF-LW. The algorithm SVF-EF seems most preferable among all priority based algorithms since it outperforms other algorithms while achieving a good fairness. For the random ordering, we see a large range of stretch values which is reflected in the maximum and standard deviation values. In the rest of the experiments the performances of the HYBRID and Block-Scheduling algorithms are compared with SVF as the best priority based algorithm. In Figure 3.6, The average completion time for the Block-Scheduling and HYBRID algorithms are compared with Random-Random and SVF-EF priority based algorithms. It can be seen that the heuristics are doing better than Random-Random algorithm considerably. This is a significant result, considering the fact that the implemented Random-Random is not a trivial procedure. In Random-Random,

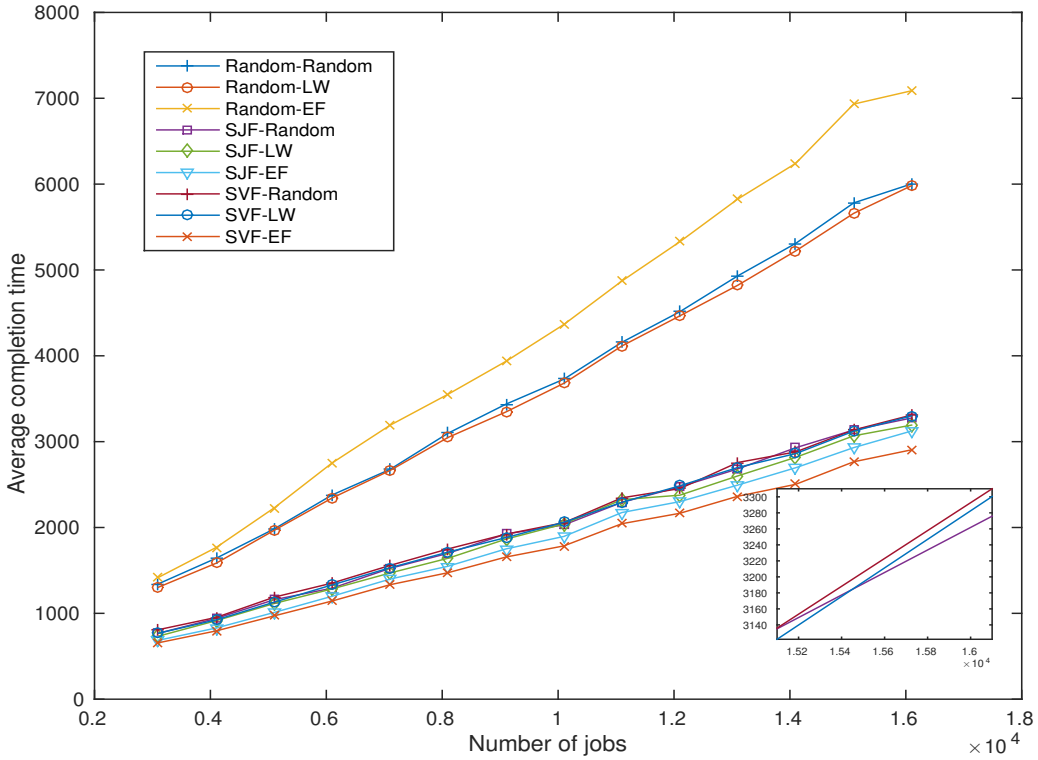

Figure 3.3: Average completion time vs. number of jobs for synthetic data set, the milder growth in average total completion time for SJF and SVF is observable. Job sizes are generated from uniform distribution.

Table 3.2: Comparison of stretch in the priority based scheduling algorithms, 5000 jobs are run on 10 parallel machines

Method	Average	Max	std
RANDOM-RANDOM	507.80	22011	1385.19
RANDOM-LW	503.00	21516	1371.94
RANDOM-EF	514.13	19056	1265.38
SJF-RANDOM	64.85	835	72.72
SJF-LW	61.03	1470	85.44
SJF-EF	**39.82**	99.86	25.49
SVF-RANDOM	83.91	2235	142.25
SVF-LW	80.83	1977	129.25
SVF-EF	**40.81**	131.08	33.63

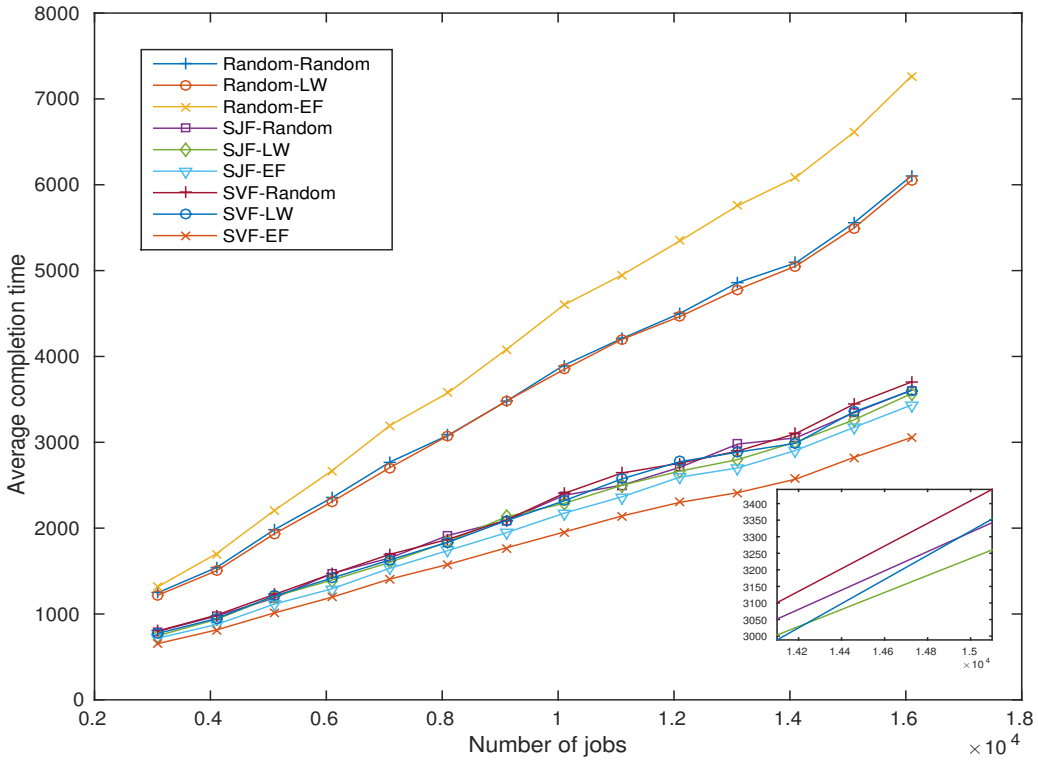

Figure 3.4: Average completion time vs. number of jobs for synthetic data set. Job sizes are generated from geometrical distribution. The growth pattern is similar to the Fig 3.3, however SVF-ef curve has a larger margin with other curves.

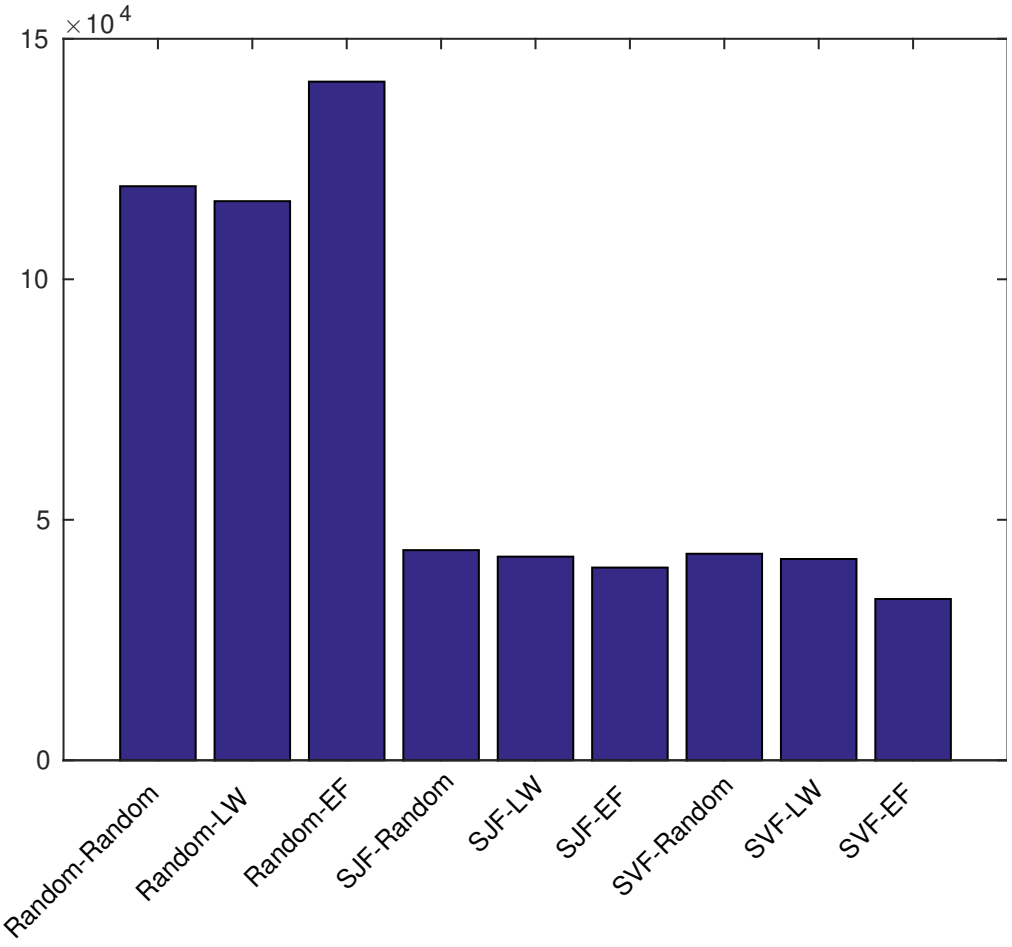

Figure 3.5: Average completion time comparison for 800 jobs sampled from HPC2N data set. SVF-EF outperforms other priority based algorithms.

in order to run a job on randomly selected server, the earliest available time that the jobs can get enough resources to run is calculated. So, although the server selection and job ordering are random, jobs are run at the earliest available time on the designated server. HYBRID is the best performing heuristic. This is in line with theoretical results in Section 3.3.

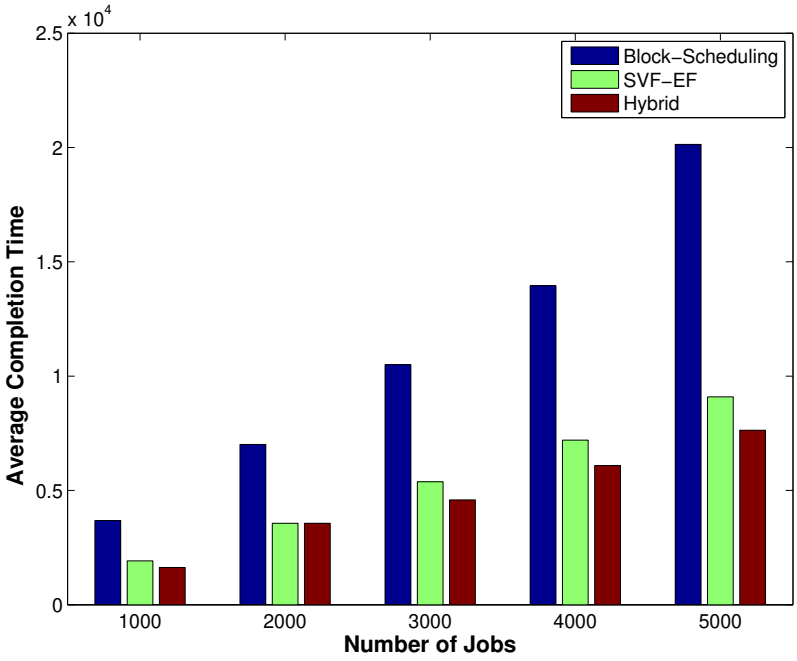

Figure 3.6: Comparison of the proposed method. HYBRID is doing slightly better than SVF-EF and Block-Scheduling is about two factor off the performance of HYBRID and SVF-EF.

The proposed constant approximation algorithms, Block-Scheduling and HYBRID, are good candidates for scheduling jobs non-preemptively on parallel machines. Both of the algorithms have good performance in terms of total completion time and they both scale well. There is a trade-off between performance and runtime as former is faster and latter has a better performance.

In terms of performance, HYBRID is the best. Block-Scheduling is about factor of two off the SVF-EF and this is a normal consequence of rounding job sizes to the next power of two. To improve the performance of Block-Scheduling, several changes in the algorithm can be helpful. One is to follow a methodology like HYBRID to divide jobs into two groups and consider different priority rules to sort each of them. Another approach is to try to consolidate the blocks that are complimentary in terms of resource usage. Another idea is improving the rounding error by categorizing jobs into finer grained groups of jobs. This may improve the objective function, but will increase the runtime.
In Figure 3.4.2, the average total completion time vs. number of servers is plotted . We see that HYBRID has the best performance.

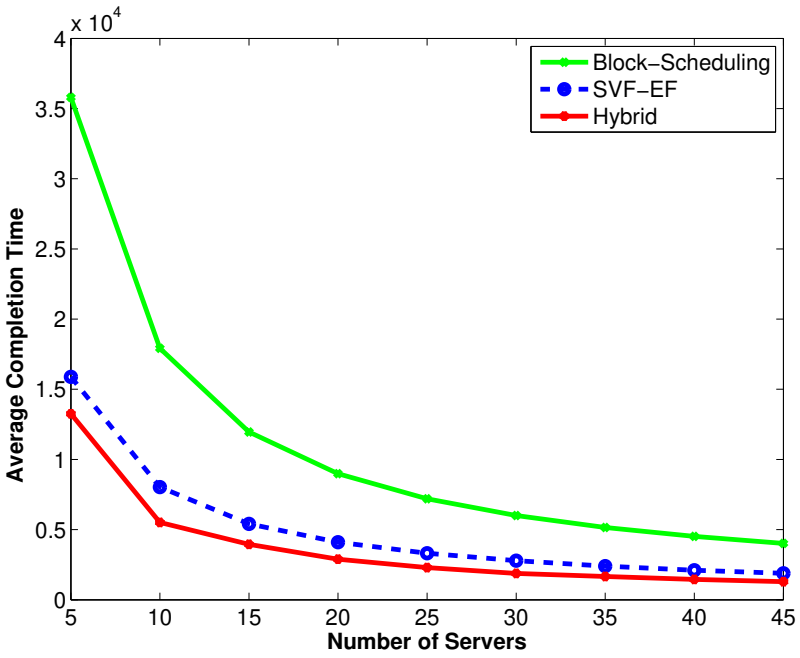

Figure 3.7: The objective function of the proposed algorithms are compared as the number of servers are increased. HYBRID has the best performance. Block-Scheduling is about two factors off the SVF-EF algorithm.

In Figure 3.4.2, the processing times of algorithms are depicted. Block-Scheduling is faster than the priority based algorithms. We have plotted its processing time in Figure 3.4.2 separately. The computational complexity of $O(n \log(n))$ makes the algorithm a good choice when there is a large number of jobs to schedule.

The SVF-EF and HYBRID are slower, as our efficient implementation of EF procedure has time complexity of $O(n)$ which leads to time complexity of $O(n^2)$ for any priority based algorithm that uses EF for machine assignment. Between the two slower algorithms, HYBRID does a little better than SVF-EF as it divides the jobs into two smaller groups which makes it a constant factor faster.

3.5 Conclusion

In this chapter we proposed new algorithms for scheduling jobs with varying demands on multiple machines without interruption. For this scheduling scenario which is widely observed in practice, we considered one of the most popular objectives, minimizing total completion time. Our algorithms are the first algorithms with performance guarantees for this specific problem to the best of our knowledge. Our work gives in-depth insights on the problem, and develops heuristics with proved approximation guarantees. In particular, we designed two simple and scalable algorithms, HYBRID and Block-Scheduling and showed why they outperforms other intuitive algorithms, which is verified by our experiments.

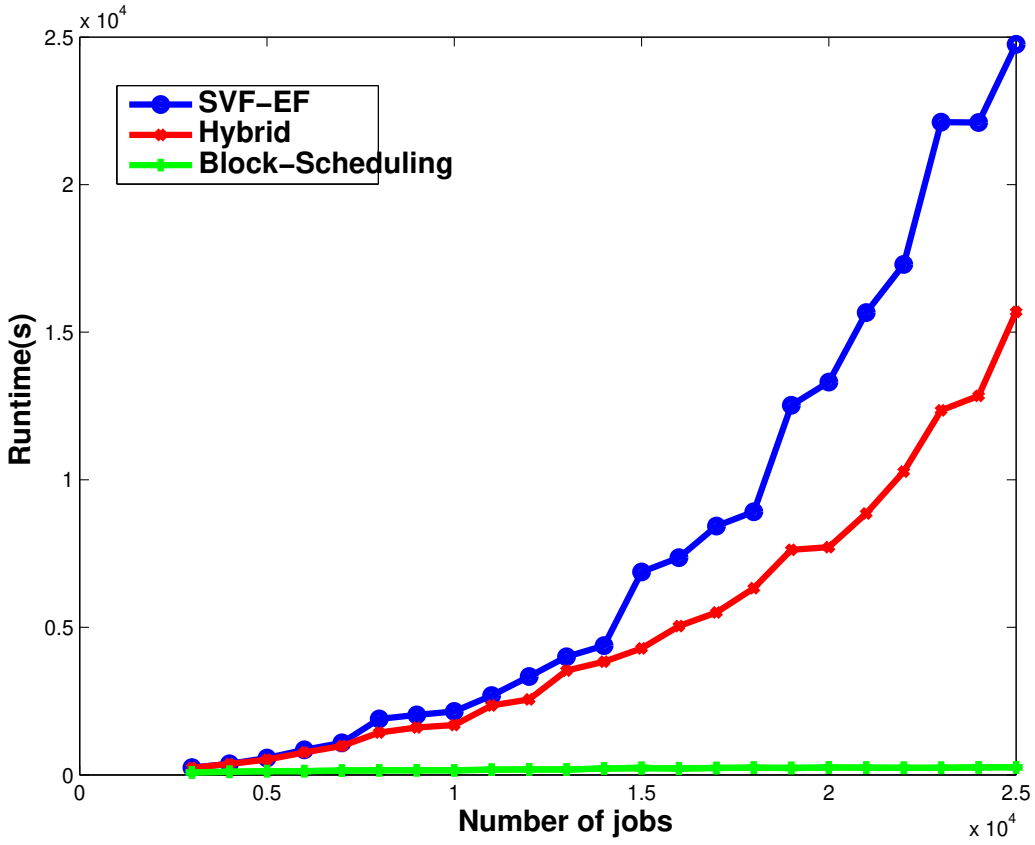

Figure 3.8: figure
The runtime of three proposed algorithms are plotted versus number of jobs. Runtimes are in seconds. It is observable that $O(n \log(n))$ time complexity of Block-Scheduling keeps its runtime negligible for scheduling of jobs up to 8000.

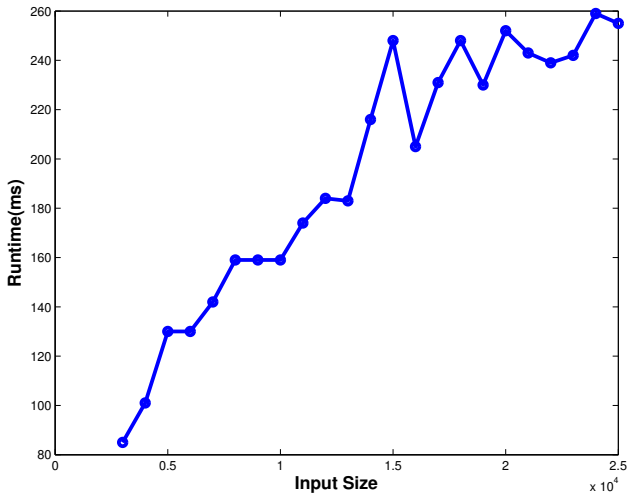

Figure 3.9: figure
The run time of Block-Scheduling algorithm is demonstrated in a wider range of up to 37000 jobs. The run times are in miliseconds.

We proposed two novel algorithms with provable approximation guarantees for scheduling jobs with non-uniform demands on multiple homogeneous servers without preemption. These two algorithms are the first constant approximation algorithms for all instances. The performance efficiency of the algorithms was studied and demonstrated via simulation experiments. Our work is only a starting point and it can be extended in many interesting directions such as considering different types of resources as well as varying demand for each resource over time and heterogeneous servers.

In another direction, our simple and efficient algorithms can be tailored for scheduling workloads on current cloud computing platforms. New levels of resource efficiency can be achieved by using our novel algorithms as scheduler that gets input data about resource usage and task runtime from workload characterization modules.

Chapter 4

Handling Inaccuracies in Scheduling HPC Applications in Cluster

The performance of scheduling algorithms for HPC jobs highly depends on the accuracy of job runtime values. Prior research has established that neither user provided runtimes nor system generated runtime predictions are accurate. We propose a new scheduling platform that performs well in spite of runtime uncertainties. The key observation that we use for building our platform is the fact that two important class of scheduling strategies (backfilling and plan-based) differ in terms of sensitivity to runtime accuracy. We first confirm this observation by performing trace-based simulations to characterize the sensitivity of different scheduling strategies to job runtime accuracy. We then apply gradient-boosting-tree-regression as a meta learning approach to estimate the reliability of the system-generated job runtimes. The estimated prediction reliability of job runtimes is then used to choose a specific class of scheduling algorithm. Our hybrid scheduling platform uses plan-based scheduling strategy for jobs with high expected runtime accuracy and backfills the remaining jobs on top of the planned jobs. While resource sharing is used to minimize fragmentation of resources, a specific ratio of CPU cores is reserved for backfilling of less predictable jobs to avoid starvation of these jobs. This ratio is adapted dynamically based on the resource requirement ratio of predictable jobs among recently submitted jobs. We perform extensive trace-driven simulations on real-world production traces to show that our hybrid scheduling platform outperforms both pure backfilling and pure plan-based scheduling algorithms.

4.1 introduction

With ever increasing usage of HPC clusters by scientific researchers, the diversity of applications submitted to HPC clusters is non-deniable. This diversity requires smarter scheduling strategies to keep resource usage efficient while meeting the clus-

ter cusomers' preferred performance goals. The management of job deployment and scheduling is performed by job management systems in HPC clusters. Job management systems for HPC clusters need to perform resource allocation and job scheduling cleverly to avoid capital and operational expenses for operating the cluster. Extensive research has been done in designing scheduling platforms with better performance and efficiency. However, most of these algorithms require an accurate value for job runtime. It is well-known that user provided runtime estimations are far from accurate (89; 140; 47). Several systems-generated runtime prediction approaches based on the runtime of previously executed jobs have been proposed. However, due to the inherent uncertainties in job runtime values, perfect prediction of runtimes is not possible (141). To the best of our knowledge, no reliability estimation for individual runtime predictions has been done in the past. Our studies on HPC trace data show that although prediction approaches predict runtimes for some jobs very accurately, they provide inaccurate prediction for a subset of submission jobs. The reliability of individual job predictions provides valuable information to be used by scheduling platforms. We use a supervised machine learning approach to estimate the reliability of each job runtime prediction. In other words, we determine if a runtime prediction for each new job is reliable using a machine learning model trained on previously completed jobs. We then use our estimations of the accuracy to design a scheduling platform that deploys an appropriate scheduling strategy based on the predictability of the runtime for each job. For that purpose, we first conduct experiments in multiple HPC trace datasets and measured sensitivity of different scheduling algorithms to job runtime accuracy. We then designed a reliability estimation approach to determine the accuracy estimates which will be a valuable input for our scheduling platform.

To be able to pick the best scheduling strategy based on the reliability of job runtime prediction, it is worthwhile to study the sensitivity of job scheduling policies to job runtime accuracy. There are two well-known groups of scheduling policies for HPC jobs: Backfilling policies and plan-based policies. Backfilling tries to fill the free resources with smaller jobs. However, FCFS-backfilling is far from optimal because the initial ordering of jobs limits the feasibility of resources for later shorter jobs. The popularity of FCFS-backfilling is due to the greedy nature of backfilling. It performs well with inaccurate runtime prediction. However, with the advent of more accurate online prediction models, more plan-based approaches have been proposed (160). One may assume that with more accurate prediction approaches, plan-based scheduling algorithms perform best. This is not true. One issue about using machine learning approaches for runtime prediction is the fact that machine learning models are trained based on minimizing a loss function. The loss function often minimizes average error over all training points and does not necessarily perform well for all data points. The question is whether we can determine if a machine learning model accurately predicts the runtime for a newly submitted job. In Subsection 4.4.4, we see that we actually can have accurate estimations of runtime prediction accuracy values for job runtimes. We call this estimation the prediction reliability estimation following some previous works in machine learning literature (21). Using the reliability estimation,

we categorize jobs into two categories of *predictable* and *unpredictable* jobs and design our Hybrid-Scheduling platform around these two categories of jobs.

The research presented in this chapter addressed three questions. (1) Do different scheduling strategies differ in sensitivity to prediction accuracy? (2) Can we estimate the prediction reliability of individual jobs? (3) Can we improve the performance and efficiency of scheduling HPC clusters with the knowledge of the answer to the first two questions? To answer the first question we performed sensitivity experiments with scheduling strategies from the two broad groups of backfilling and plan-based. We observed that backfilling strategies are less sensitive to runtime accuracy. We also observed that the performance of plan-based strategies significantly improves with more accurate runtime values. We then studied prediction reliability and experimented existing approaches in machine learning literature for estimating prediction reliability for job runtime values. Based on our experiments we designed a supervised model that estimates the accuracy of runtime using gradient boosting tree regression. The Pearson correlation for our estimated accuracies and the actual accuracy is 0.84 on our test data. We then designed a hybrid scheduling platform that picks a policy for scheduling each job based on the reliability of its runtime prediction. Our platform first schedules the predictable jobs using a plan-based scheduling strategy and then backfills the less predictable jobs on top. To avoid starvation, a dynamically adaptive portion of resources are reserved for backfilling jobs. However, backfilled jobs can use all remaining available resources to avoid resource fragmentation. Our extensive trace-based simulations show that our platform outperforms each of the two strategies deployed individually in terms of performance and efficiency.

We start by presenting the problem and related background in Section 4.2. We review related work for HPC scheduling as well as prediction reliability estimation in machine learning literature in Section 4.3. We present our approach for reliability estimation and Hybrid Scheduling Platform in Section 4.4. We present our trace-based experiments with actual data in Section 4.5. We conclude our work in Section 4.6.

4.2 Background and Problem Description

In this chapter, we are interested in scheduling jobs in HPC clusters. The HPC applications are submitted to a central Application Management System. The job management system decides the allocation of resources to the applications. As the resources are finite, the applications may need to wait until they acquire resources. The users are asked to provide running time and the required CPU and memory for their applications. It is well known that the users submit an overestimated runtime mostly because the application manager kills the applications if they take longer than the user estimated duration (71).

In order to better present the problem, we first define our notion of a job in HPC cluster: a *job* or *application* j is considered with specific submission time and resource requirement. At the time of submission, a value of runtime and resource requirement is submitted by the user. There are n independent jobs (indexed by integers), where

application j has the following characteristic:

- Submission time: r_j
- Resource requirement: d_j
- Actual running time: p_j
- Requested running time: $\hat{p}_j$.
- Additional features (descriptors) including the user that submitted the application, the time of the day the application submitted, etc.

In Fig 4.1, an application j is illustrated as a solid rectangle, with length equal to actual runtime (p_j) and resource requirement equal to d_j. In the right figure, the abstraction of resources in the HPC cluster is illustrated. The vertical dimension represents the total resources in the system and the horizontal axis denotes time. In this section, we first present a background on scheduling approaches for HPC clusters and introduce the issue of uncertainty in HPC workload traces. We then present the formulation of the problem we study in this chapter.

4.2.1 Common Scheduling Algorithms for HPC Workloads

FCFS (First Come First Served) is the most well-known scheduling algorithms for HPC jobs. FCFS schedules jobs in order of their submission. FCFS is a list scheduling algorithm that prioritizes jobs based on their submission time. In list scheduling algorithms, also known as queue-based scheduling algorithms, if there are enough available resources, resources are allocated to the submitted job and the job starts to process. Otherwise, the job is kept in a queue. Using FCFS does not consider the geometry of jobs to pack them tightly into resources. To improve the performance of FCFS, two strategies have been proposed: Backfilling (FCFS-BF) (135) and Plan-Based scheduling (160) algorithms. In Fig 4.2, an FCFS-Backfilling scheduling strategy is compared with a plan-based scheduling algorithm. While Backfilling backfills the free resources available after FCFS scheduling with smaller jobs from the back of the queue, plan-based scheduling uses the runtime of jobs in the queue to find the near optimal ordering jobs before assigning the jobs. As these two groups of scheduling algorithms are the building blocks of our hybrid scheduling platform, we elaborate their characteristics in the following subsections.

FCFS Scheduling with Backfilling

In FCFS with backfilling, jobs are prioritized based on their submission time to the system. In FCFS with backfilling, a rule is used to select some jobs from the back of the waiting queue to run earlier. Several variety of backfilling algorithms are proposed including easy (128), conservative (135) and slack backfilling (138) algorithms.

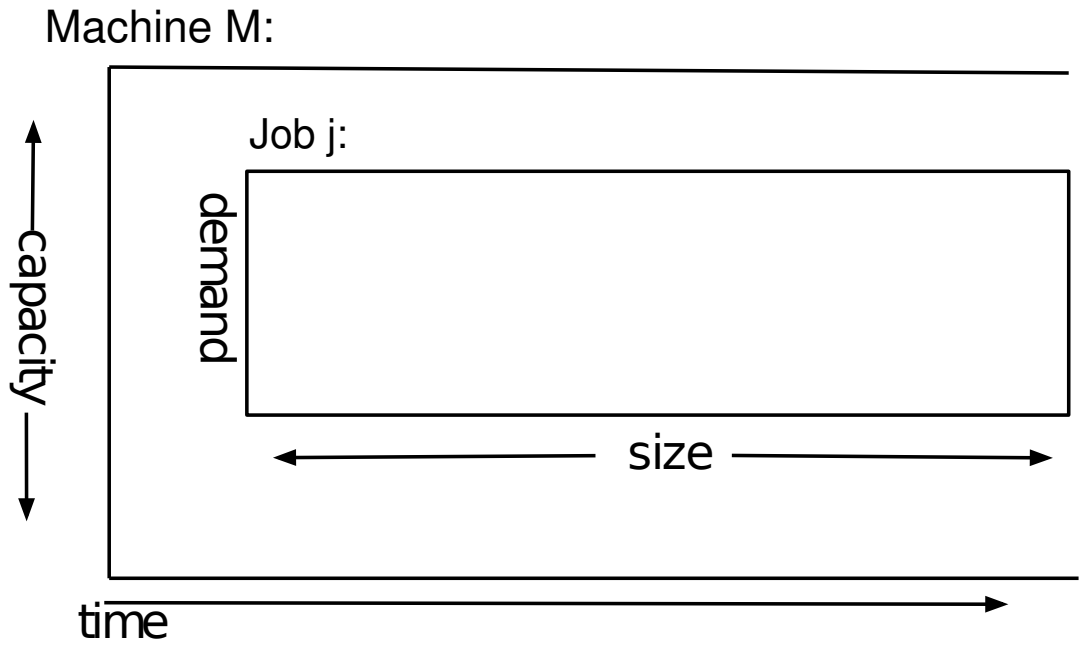

Figure 4.1: Each job j is illustrated as a two-dimensional rectangle with height equal to its demand d_j and width equal to its size p_j. Each server has a unit capacity. Jobs can run simultaneously on each server as long as the total demand/height of running jobs do not exceed the server's capacity.

Although FCFS only rely on submission time to order the jobs, all these backfilling approaches rely on runtime values to make the backfilling decision. Most of the resource management systems deployed in HPC clusters including SLURM (154), Cobalt (38), IBM LoadLeveler (128) use FCFS with backfilling. FCFS scheduling algorithms are used mainly due to their simplicity and scalability as well as stability to inaccurate input runtimes. The easy-backfilling algorithm acts like a greedy first fit scheduler in the case that the next job in the queue has more resource demand than the available resources. It takes the first job from the back of the queue that fits into the available space.

One important observation is that as EASY-backfilling tries to backfill jobs greedily into available holes created by the FCFS ordering of jobs, its performance does not degrade substantially with inaccurate runtime estimates. On the other hand, the performance of the EASY does not improve substantially with more accurate runtime values. FCFS-SJF was proposed in (142) to use the application runtime for backfilling decision. In FCFS-SJF, the backfilled jobs are chosen in the order of increasing runtime. FCFS-SJF is commonly used in the works that propose more accurate prediction approaches to runtime prediction as SJF-BF is more sensitive to runtime prediction accuracy than EASY. We will also consider FCFS-SVF that orders jobs based on volume(multiplication of runtime and required CPU). FCFS-SJF and FCFS-SVF have some level of sensitivity to runtime accuracy, but still have acceptable performance in absence of accurate runtime prediction. In our experiments we use FCFS-SJF and FCFS-SVF as representatives of FCFS with backfilling algorithms.

Plan-based Scheduling Algorithms

On the other side of the scheduling algorithms spectrum are the plan-based algorithms. Instead of deploying jobs immediately, plan-based approaches make a scheduling plan-based on a group of submitted jobs. They try to find a near optimal ordering of jobs to optimize scheduling performance. The main issue with plan-based scheduling algorithms is the fact that their performance is highly sensitive to the accuracy of jobs' runtime predictions. As in real-world scenarios, user runtime estimates used for scheduling are not accurate, plan-based scheduling algorithms do not perform well. We study several plan-based and backfilling approaches and propose an adaptive hybrid scheduling platform. Our sensitivity analysis experiments in the next subsection show how plan-based work well with accurate and backfilling with inaccurate predictions. Plan-based scheduling algorithms try to search over the solution space to make the best scheduling decision for each job. As the problem is dynamic and jobs are submitted over time, the planning routine needs to be done periodically and based on the jobs already in the systems. Several plan-based scheduling policies have been proposed. Some policies propose a complete search over the solution space, and some propose local search to improve the computation overhead (160; 152; 82).

Although these methods claim to have better performance than backfilling scheduling methods, several issues make them less popular for cluster managers. First, for most of these approaches, the computational the overhead for decision making makes

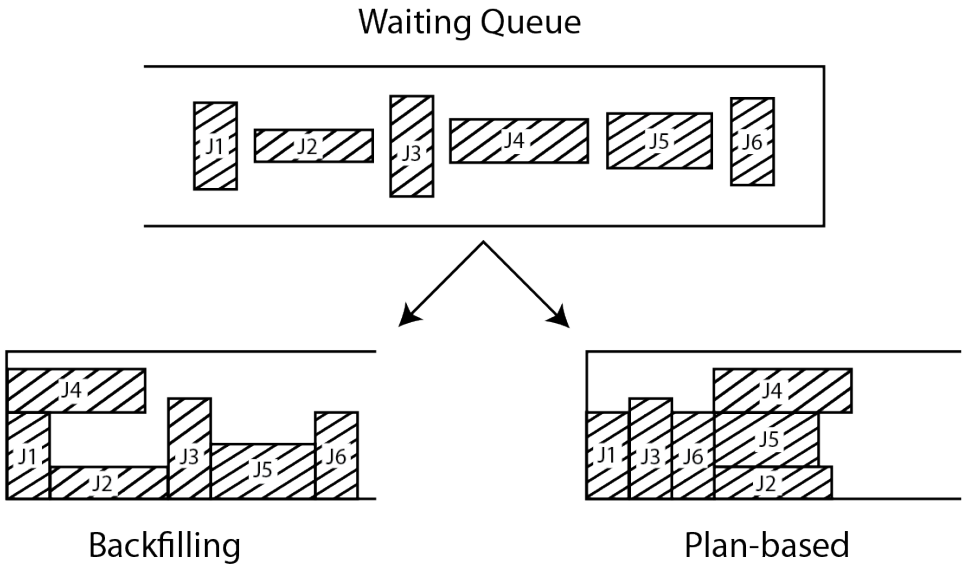

Figure 4.2: The comparison of a plan-based scheduling algorithm (online-SJF) and a backfilling scheduling algorithm (FCFS-SJF) algorithms on an example of seven jobs.

them less favorable. Furthermore, they completely rely on job runtimes for their decisions and perform poorly if the runtimes are not accurate.

In this chapter, for the choice of plan-based strategy in our hybrid scheduling platform, we use online priority-based scheduling algorithms. Online priority-based scheduling algorithms are the plan-based version of commonly used priority-based scheduling algorithms. Online priority-based scheduling algorithms have the high performance of plan-based scheduling algorithms on the availability of accurate runtime values while they have a low computational overhead. Shortest Job First (SJF) algorithm is well known to have optimal performance in the offline case when one job is allowed to run at a time. In (Im et al.) authors proposed Smallest Volume First (SVF) as the two-dimensional extension of SJF that reaches near-optimal performance in offline case. In comparison with other plan-based scheduling including heuristic search algorithms, list-based scheduling algorithms are considerably faster. Online-SJF and online-SVF are dynamic. Similar to other plan-based scheduling algorithms, new ordering is computed at the time of the system events. With using appropriate data structures like Heap, reordering the list takes constant time. Our experiments show that SVF outperforms SJF and other commonly used scheduling algorithms when accurate runtimes are provided. In this work, we will use online-SVF as the plan-based scheduling algorithm and call it Plan-Based scheduling in the rest of this chapter.

4.2.2 Sensitivity to Job Runtime Accuracy

To study the sensitivity of backfilling and plan-based schedulers, we studied four traces from Parallel Workload Dataset (wor) in detail. These datasets contain traces of applications submitted to HPC clusters. The first observation as noted by several previous works (142; 133) is inaccurate user estimates of job runtimes. In recent years, several prediction approaches are proposed (142), (55) and some of them can predict application runtimes as accurate as 80%. The question is how these more accurate predictions affect the scheduling performance both for customer satisfaction criterion (performance) and cluster provider criterion (utilization). To answer this question, we perturbed the available traces to provide traces with variable runtime prediction accuracy and simulated several scheduling algorithms discussed in the previous section to compare the scheduling algorithms regarding their sensitivity to accuracy. We also want to compare how different scheduling algorithms perform with more accurate runtimes. FCFS does not use runtime estimates and schedules jobs on the order of submission, so the average wait time of FCFS is not impacted by more accurate runtime prediction. The important point to notice is the sensitivity of online-SJF and online-SVF to the accuracy of runtime. These two algorithms do not perform well with user-estimated runtimes which are generally inaccurate.

Through our extensive trace-based simulations, we realized that plan-based scheduling algorithms are more sensitive to prediction accuracy than backfilling algorithms. The good news is that they outperform backfilling algorithms if an almost accurate prediction is available (accuracy more than 60%). We perturbed the existing traces from ANL, LLNL, HPC2N, and SDSC to build 20%, 40%, 60% and 80% runtime value accuracies and simulated different backfilling and plan-based scheduling algorithms with the perturbed runtimes. The results of these experiments are demonstrated in Fig 4.3. We can see that for all four datasets, for accuracy level higher than a threshold (40% to 80%), online-SVF outperforms all the other algorithms in terms of wait time.

4.2.3 Job Runtime Prediction Reliability Estimation

It is well-known that provided runtime values by users are far from accurate (142). Several approaches have been proposed to generate more accurate runtime predictions based on available information about the job at the time of submission. In (133), several time-series-based approaches are used to predict job runtimes. A machine learning approach was proposed in (55) where the authors propose an online learning model that makes a prediction for each newly submitted job and updates the prediction model with newly available runtime after completion of each job. However, to the best of our knowledge, there has not been any study about confidence or reliability of runtime predictions for individual jobs.One important issue in using predicted runtimes is the fact that for some jobs, the prediction accuracy is low. We need to measure the expected prediction error for individual jobs. This can be done by estimating individual prediction reliability.

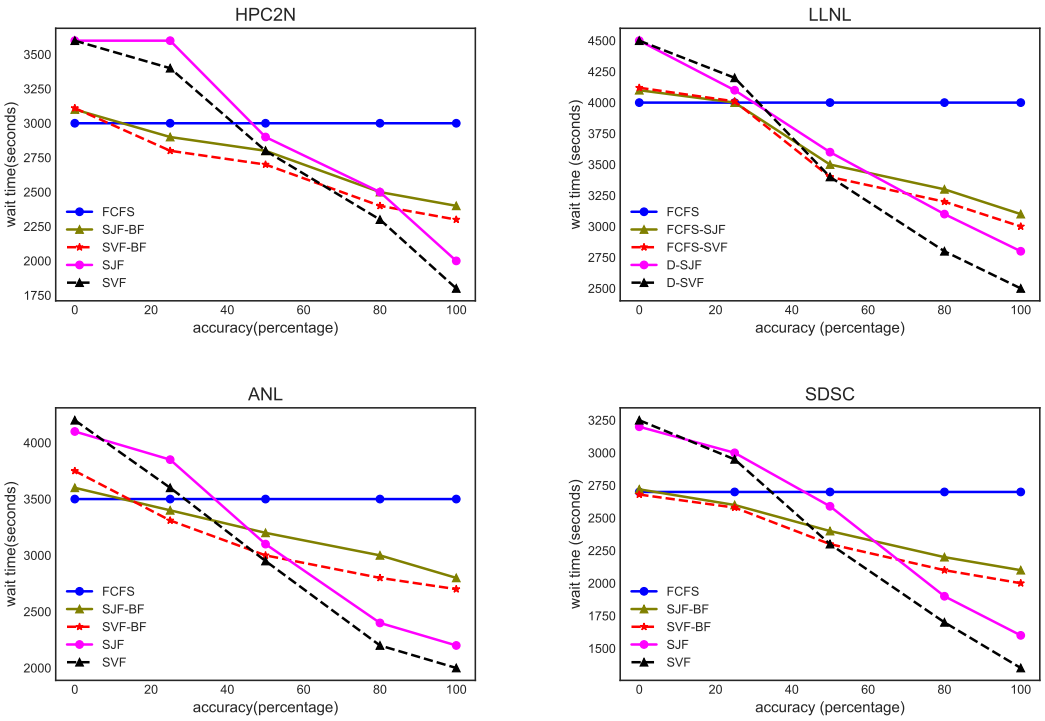

Figure 4.3: Wait-times of FCFS, FCFS-SJF, FCFS-SVF, Online-SJF, and Online-SVF are plotted for traces with different accuracy levels.

Evaluating the quality of prediction uncertainties is a challenging problem in machine learning (87).

Using the individual prediction accuracy metric (4.2) for previous jobs, we train a regression model to predict job runtime accuracy for the newly submitted jobs. When a job is submitted to the HPC cluster, information including the user id, time of submission, requested runtime, CPU and memory is available for each job, we extract useful features from job log. We explain the details of our approach to Section 4.4. The prediction accuracies estimated by our reliability estimation machine have a Pearson correlation of 0.84 with actual accuracies for HPC2N trace data.

4.2.4 Formulation of the Problem

The problem studied in this work is to execute a set of concurrent parallel jobs with rigid resource requirements on an HPC platform with m units of resources. The jobs are submitted over time in an online manner.

The resource requirement d_j of an application j is equivalent to the user requested resource known at the time of submission. The actual value of runtime is only known a posteriori when the job completes. The problem we are trying to solve is how to schedule applications to achieve better performance and utilization, without knowing the accurate runtime values at submission time.

Although application runtime has some correlation with several features as discussed in previous works (55), 100% accurate prediction of application runtimes is not possible (55). Thus, we are looking for scheduling solutions that improve the performance and utilization over the currently used approaches when 100% accurate prediction is not available for all jobs in the system. More specifically we look for a scheduling solution that:

- is online.
- has low overhead.
- performs well even with inaccurate runtime predictions.
- achieves high performance when accurate predictions are available.
- outperforms commonly used scheduling strategies in a real-world setting (accurate runtime is available for a subset of jobs).

4.3 Related Work

4.3.1 Estimating Prediction Reliability

Estimating prediction reliability refers to estimating how well a prediction model will perform on unseen data. In this work, we specifically focus on estimating prediction reliability for individual out of sample data points. There have been several works in machine learning literature on determining the reliability of the prediction model for individual data points (53), (123). These approaches assume that features and target values are generated independently from the same probability distribution and they calculate confidence for predicted target values using p-value measure. In (21), authors propose local regression sensitivity analysis to provide prediction reliability values. The application of their approaches requires multiple runs of the prediction step to determine the variance of the results and is not appropriate for our purpose of estimating accuracy values on-the-fly. Some other existing approaches are designed for a special group of prediction models and cannot be prescribed to estimate reliability for an arbitrary prediction model (110; 149; 87; 79).

Authors in (119) have reviewed prediction uncertainty for online prediction problem. They compared multiple reliability estimation using the correlation coefficient of estimated accuracy and the actual accuracy for the prediction model. In their work, the similarity-based reliability estimation approach is implemented using K-nearest-neighbors (KNN) prediction approach and is shown to be a well-performing estimation approach. Similarity-based reliability prediction approaches consider the accuracy of previous predictions of the same prediction model for similar examples in the input space. A more general approach for similarity based prediction reliability estimation is to train a regression model that maps feature space and predicted target value to corresponding accuracy. Using regression models to estimate prediction

accuracy has been proposed in the past (52; 143; 14). Building a regression model helps to map feature set characteristics to prediction accuracy and helps to identify the characteristics of features that affect the performance of the classifier. In our work, we adopt a similarity-based approach for reliability estimation. We use gradient boosting tree regression instead of K-nearest neighbor as accuracy prediction model and showcase its better estimation performance in comparison to the K-NN model in Section 4.4.

Estimating prediction accuracy has often been proposed to help model selection or iterative model improvement to achieve better prediction accuracy. However, in our work, the goal of prediction reliability estimation is whether to rely on predicted target values or not. If estimated prediction accuracy for a data point is too low, our scheduling platform ignores the predicted runtime and uses the user requested runtime with a less runtime accuracy sensitive scheduling policy.

4.3.2 HPC Scheduling and Runtime Uncertainty

The necessity of runtime prediction for parallel applications has been highlighted since the last years of 20th century (58). Different approaches have been proposed to predict HPC application runtimes with different machine learning methodology as well as prediction features inputs (142; 139). Several time-series based methodologies are proposed to use information available about completed jobs in the system to predict the runtime for newly submitted jobs. They are mainly exponential smoothing and moving average methodologies that predict future values based on the recent runtime values. As noted by (133), these methods are not accurate and will not improve scheduling performance and utilization significantly. Some earlier works profile the existing applications and form a model similar to the regression decision tree to predict the runtime of the new application based on the most similar application profile (129; 58). (129) applies genetic algorithms to search among the history logs to find the most similar attributes. Several statistical methods fit a distribution to previous data and predict the new job runtime with mean and confidence interval of the inferred distribution (142). (109) draws sixteen different distribution from previous data and designs a hidden Markov Model to transit along these sixteen states. Some other researchers have considered additional features for each job in trace and perform prediction based on the similarity between these features of the jobs (102; 125). Some more recent works focus on the specific family of scientific workflows and use machine learning for predicting runtime and resource usage for these applications (100). Several works have proposed interrogating the codes to extract features for runtime prediction. This is also not practical in many cases due to privacy considerations. All these approaches are static, meaning that they train a static model based on the available applications and use the model for prediction the runtimes of new applications. Applying such a static model for a dynamic environment like cloud leads to inaccurate predictions.

Making an accurate prediction for application runtime is not easy. In fact, the scarcity of relevant training data makes the model building cumbersome. Several online prediction methods have been proposed that use the recently completed jobs in the same trace to strengthen the predicting power of their model (133), (139).
Some authors correctly identified the importance of avoiding overfitting the value of runtimes and considered approaches that favor under-prediction over over-prediction as over-predicted runtime values impose the expense of killing incomplete jobs (47; 55).
Similar to our work, some previous works considered inaccurate predictions. They proposed a solution to handle inaccuracies in predicted runtime values. Authors in (40), considered error margins for runtime prediction to be considered for runtime adjustment. In (117), authors developed an execution delay model for runtime prediction and designed an adaptive stochastic allocation strategy for production workload traces.

4.4 Proposed Hybrid Scheduling Platform

The study of sensitivity in Section 4.2.2 gave us an intuition to design a hybrid scheduling platform. Since in real-world situations, accurate values of job runtimes are not available, we propose a Hybrid Scheduling platform that uses FCFS with backfilling for jobs with unpredictable runtime. At the time of job submission, the normalized regression model is used to predict the job runtime value in an online manner. Then, jobs are classified into two queues of *predictable* and *unpredictable* using our trained reliability estimation model. Our platform schedules jobs with higher expected runtime accuracy using plan-based scheduling. Our platform then performs backfilling to schedule the jobs in the *unpredictable* queue on the remaining available resources. The *predictable* and *unpredictable* jobs are defined below.
Predictable Jobs: These are jobs with high prediction reliability. We characterize these jobs with estimated runtime accuracy of 60% or more.
unpredictable Jobs: These are jobs with low prediction reliability. We characterize these jobs with estimated runtime accuracy of less than 60%.
Our resource manager dynamically determines the resource quota for pure priority based algorithm based on the ratio of predictable jobs.

4.4.1 Proposed Design

In Subsection 4.2.2, we demonstrated how priority-based scheduling algorithms outperform backfilling algorithms when we have an almost accurate (accuracy of 60% and higher), prediction of job runtimes. We use this observation to design a hybrid scheduling algorithm. Similar to other algorithms discussed in Subsection 4.2.1, our algorithm is online, and scheduling takes place in rounds. Two different policies, SVF and FCFS-BF, are performed at each round. Using our reliability prediction model, we know for which subset of jobs we have more accurate runtime values. Our hy-

brid model will apply Plan-Based scheduling for the subset of jobs and backfills the remaining jobs on the top. We propose a hybrid scheduling algorithm that smartly combines the backfilling algorithms and plan-based scheduling algorithms. With our hybrid design, we will benefit from the effectiveness of backfilling scheduling approach when the runtime predictions are inaccurate and we will benefit from the near optimal performance of plan-based scheduling when the predictions are more accurate. We determine the reliability of predicted runtimes using a supervised machine learning method of gradient boosting trees as presented in Subsection 4.4.4. The supervised regression outputs an estimation of the prediction accuracy. Our scheduling platform puts jobs in two queues of *predictable* and *unpredictable* based on their estimations of accuracy. For the jobs in *unpredictable* queue, the platform does not rely on system-generated predictions. The platform uses the runtimes requested by the user to backfill *unpredictable* jobs. Our hybrid scheduling platform assigns a subset of resources to plan-based scheduling to improve utilization of the resources and tops the remaining unused resources with backfilling. Our experiments prove the better performance and CPU utilization of the proposed hybrid approach to both pure plan-based and pure backfilling methodologies. Figure 4.4 presents the Hybrid-scheduling platform

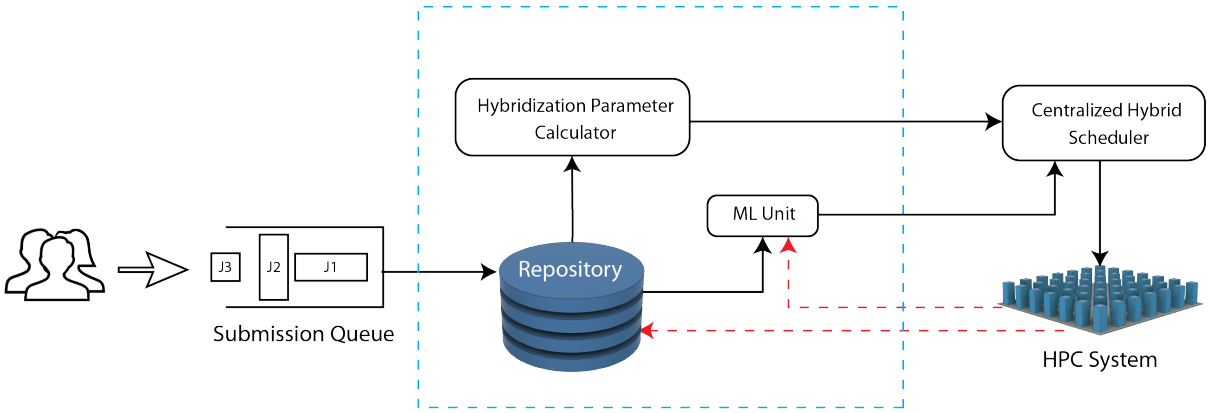

Figure 4.4: Overview of the HS platform design.

in detail. As demonstrated in Figure 4.4, when a job is submitted to the cluster, its information is stored in the repository. ML-unit calculates the runtime prediction as well as estimated prediction accuracy. The hybridization parameter calculator calculates the hybridization parameter based on the estimated prediction accuracy values as explained in Section 4.4.3. The central scheduler uses the predicted runtime to order *predictable* jobs on α portion of resources and backfills the *unpredictable* jobs on top. After each job completes on the HPC system, its actual runtime is recorded in the repository.

As demonstrated in Fig 4.4, our scheduling platform has four main components: ML-unit, hybridization parameter adjusting unit, centralized scheduler, and repository. When a job is submitted to the system, its runtime and prediction accuracy

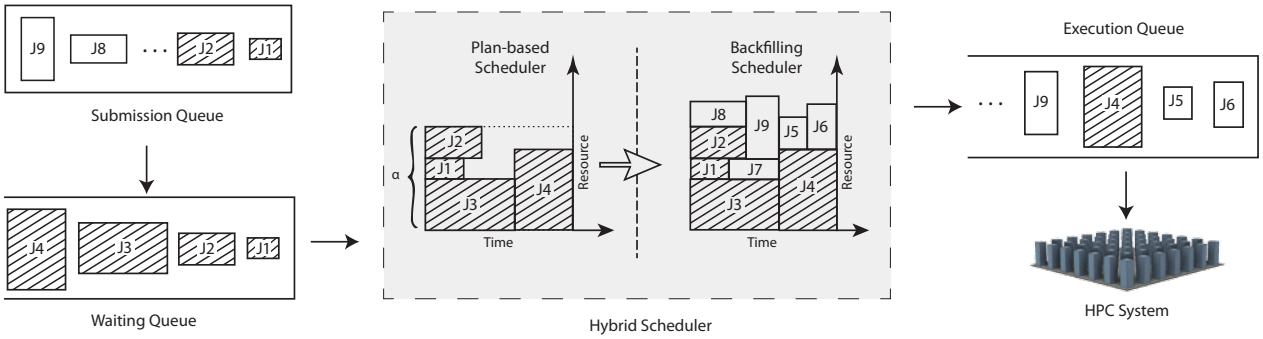

Figure 4.5: Central scheduler design.

estimation are calculated in ML-unit. The values of runtime prediction and prediction accuracy estimation are used by the central scheduler to determine the deployment time of the new job in the HPC system. After the job completes running, information about actual runtimes is saved in the repository to be used by ML-unit and hybridization parameter estimator unit. The repository contains necessary data including prediction model parameters for ML-unit as predictability estimation model. As a job completes running on HPC cluster, the runtime value is added to the repository. Centralized scheduler, hybridization parameter adjusting unit and Ml-unit are elaborated in the following subsections.

4.4.2 Central Scheduler

Using the input from ML-unit, jobs are partitioned into two queues: *predictable* and *unpredictable* jobs as defined in the beginning of current section. Jobs that are characterized as *predictable* by ML-unit are input to a new queue called *predictable*. As shown in Fig 4.5, the Hybrid scheduler is composed of two schedulers: plan-based scheduler and backfilling scheduler. At first, the central scheduler performs plan-based scheduling. It determines the starting time of each *predictable* job in the waiting queue following a plan-based scheduling algorithm as explained in Section 4.2.1. For this plan-based scheduling, only specific portion of resources using the specific portion of CPU cores determined by hybridization parameter unit is considered. Second, after the deployment plan of the predictable jobs is determined, FCFS with backfilling scheduler determines the starting time of *unpredictable* jobs on the remaining available resources. The complete schedule for all jobs is used to deploy jobs on the HPC cluster. This procedure is repeated at the time of each event in the computation cluster: Job submission, Job completion or job termination.

4.4.3 Hybridization Parameter Adjusting Unit

The ratio of resources used by plan-based scheduling policy to schedule *predictable* jobs, α, is called hybridization parameter and will be updated by hybridization parameter adjusting unit in the platform. The parameter adjusting unit updates the value of hybridization parameter, α based on the ratio of the sum of resource usage of predictable jobs to the total resource request all the jobs. The formula for updating α is presented below.

$$\alpha_{t+1} = \alpha_t * \frac{\sum_{i \in predictable} d_i}{\sum_{j \in alljobs} d_j} \tag{4.1}$$

Where d_j is the resource requirement for the job j as defined in Section 4.2. The initial value of α denoted as α_0 is chosen through a grid search as illustrated in Fig 4.13.

4.4.4 ML-unit

ML-unit is responsible for predicting runtimes and estimating the accuracy of the predicted runtimes. The runtime prediction is performed using online normalized regression model (67), (120) similar to (55). Determining the prediction reliability of jobs: To determine the prediction reliability of jobs, we use a gradient boosting tree model. The model parameters are saved in the repository and the model is rebuilt every 24 hour. One important function of Ml-unit is to determine if the jobs belong to *predictable* or *unpredictable* class of jobs. In order to study prediction accuracy for individual jobs, we need to use an appropriate metric to measure the accuracy of prediction for individual jobs.

Measure of Prediction Accuracy: Common measures of accuracy in machine learning such as Mean Squared Error (MSE), measure the accuracy of prediction globally. As we are interested in measuring and characterizing accuracy for each individual point in data set, we use a point-wise measure of accuracy that has been proposed in the literature of HPC job runtime prediction for HPC clusters (142; 133). This metric measures accuracy of runtime prediction for each job by comparing the predicted job runtime value ($\hat{p}_j$) and actual job runtime (p_j) as presented in Equation 4.2.

$$accuracy = \begin{cases} 1 & if \quad \hat{p}_j = p_j \\ \frac{\hat{p}_j}{p_j} & if \quad \hat{p}_j < p_j \\ \frac{p_j}{\hat{p}_j} & if \quad \hat{p}_j > p_j \end{cases} \tag{4.2}$$

Online Job Runtime prediction

The goal of online job runtime prediction module is to predict job runtimes in an online manner. As a job is submitted to the systems, a minimal set of features are extracted from the job description as well as resource management systems. These

feature group	feature names	number of features
user requested runtime	$\tilde{p}_j$ (reqTime)	1
actual runtime and CPU for the previously completed jobs from the same user	last, beflast, beflast2, lastcpu	4
maximum runtime and cpu for the previously completed jobs from the same user	maxrt, maxcpu	2
seasonality features	tod1, tod2, dow1, dow2	4
average and standard deviation of runtime and CPU of the jobs from the same user	meanrt, stdrt, meancpu, stdcpu	4
the number of completed jobs from the same user	prevuser	1

Table 4.1: Features considered for our prediction reliability estimation approach.

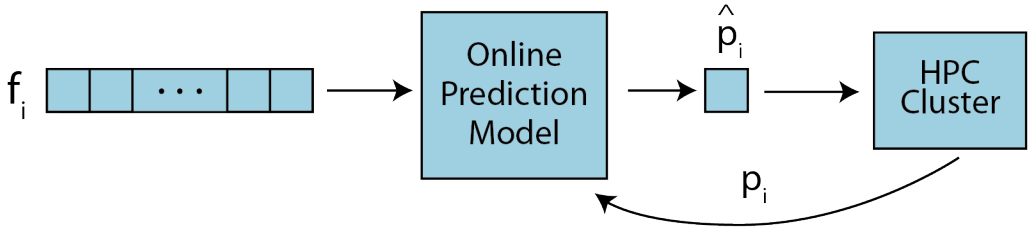

Figure 4.6: Online learning module predicts runtime for the current job based on the feedback from previous jobs.

features are used by an online job runtime prediction module to predict job runtimes. In our platform we implemented an l_2 regularized polynomial model similar to (55). New values of $\hat{p_i}$ are predicted using features available from $i-1$ completed jobs : $Z = \{z_{11}...z_{1i-1}, ..., z_{ki-1}\}$. In this work, we use online normalized regression model (120) similar to (55). Online normalized regression uses stochastic gradient descent approaches to find a polynomials function of k input features corresponding $i-1$ previously completed jobs $Z = \{z_{11}...z_{1i-1}, ..., z_{ki-1}\}$. The polynomial function is in the form of:

$$f(w, z) = w^T \phi(z). \tag{4.3}$$

The minimization of regularized cumulative loss for up to i-th completed job is:

$$arg_w \min \sum_{j=1}^{i} L(x_j, f(w, z), p_j) + \lambda \|w\|. \tag{4.4}$$

where λ is the regularization parameter and $f(w, z_j)$ is our prediction for runtime of job j.

The loss function is:

$$L(z_j, f(z_j), p_j) = \begin{cases} \lambda.f(z_j) - p_j)^2 & iff(z_j) \geq p_j \\ \lambda.(p_j - f(z_j))^2 & iff(z_j) < p_j \end{cases} \tag{4.5}$$

To focus on the symmetric inaccuracy of prediction, we consider the symmetric loss function instead asymmetric loss function in (55) and penalize over-prediction and under-prediction equally.

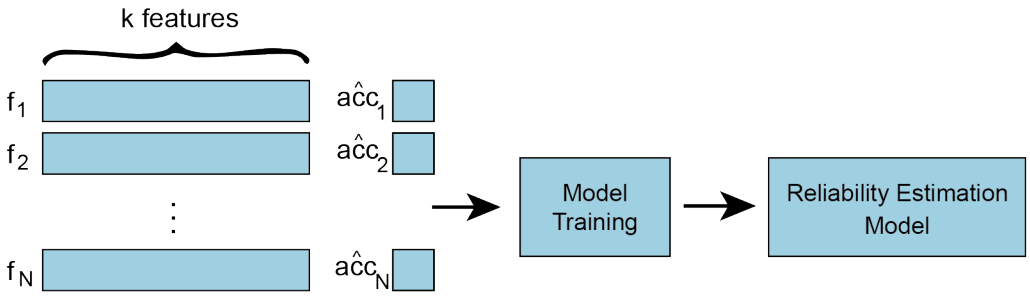

Figure 4.7: Prediction reliability estimation machine is trained using features of completed jobs $f_i = \{z_{1i}, .., z_{ki}\}$ and their corresponding prediction accuracy values acc_i.

A Meta Learning Approach to Estimate Runtime Prediction Accuracy

Meta-learning, or learning to learn, is the science of systematically observing how different machine learning approaches perform (145). Here, we are interested in estimating the prediction accuracy of our online algorithm that predicts job runtime. To estimate the prediction accuracy of job runtimes (prediction reliability), we use a supervised regression model. The regression model predicts runtime accuracy of newly submitted jobs using information about features and prediction accuracy of completed jobs. As shown in Figure 4.7, feature vectors of completed jobs, $f_i = \{z_{1i}, .., z_{ki}\}$, and their corresponding prediction accuracy, acc_i are used to train a regression model. The regression model maps the features and predicted runtime to accuracy values. We used Gradient Boosting Tree regression (50) as it is known to find non-linear mapping from data to target values (30). The trained prediction reliability estimation model is used to estimate the runtime prediction accuracy of newly submitted jobs. In Figure 4.8, the prediction reliability outputs accuracy estimation value for job i, $a\hat{c}c_i$, using feature vectors $f_i = \{z_{1i}, .., z_{ki}, \hat{p}_i\}$.

Gradient Boosting Tree regressor is a prediction approach that ensembles regression decision trees with gradient boosting (50). In gradient boosting, a model is built in a stage-wise fashion. In each stage, the existing model is boosted by optimization of an arbitrary differentiable loss function. In the gradient boosting algorithm, in each stage a tree is built based on the residuals from the tree in the previous stage. Basically, in each stage, a new gradient tree is fitted into residual values. It is important to note that at each stage, randomized samples of training data are chosen without substitution to avoid the risk of overfitting (51). Algorithm 2 shows the procedure to find gradient boosting regression tree predictor $met\tilde{a} - 2$ for input F. The algorithm has

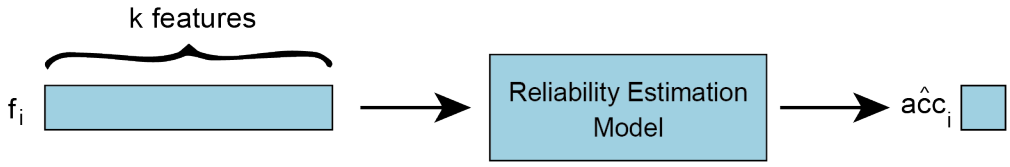

Figure 4.8: The trained reliability estimation machine is used to predict the accuracy for newly submitted jobs.

training set of k dimensional input features of size N denoted as $F = \{f_1, f_2, ..., f_N\}$ and their corresponding target accuracy values $ACC = \{acc_1, .., acc_N\}$. As the first step, a decision tree $meta_0(F)$ is fitted to F and ACC. Namely, the features to branch upon at each level of the tree as well as the threshold to make the decision is found by solving an optimization problem (66). Using the trained decision tree, one is able to find a decision region of feature space a new feature set belonging to a new data point resides. The predicted target value for a new data point is predicted by averaging the runtime of the applications in the same subregion. Then, in each stage the gradient of the residuals of predictions are calculated and a regression tree is fitted to this gradient to find R_{jm} decision regions. After calculating the multiplier for each gradient residual subtree and adding the weighted sum of subtrees to the previous regression tree, the stage is complete. The routine is repeated for a tunable number of stages.

Algorithm 2 Gradient Boosting Tree as the Meta Learning Approach

STATE Initialize $meta_0 = arg\min \sum_{i=1}^{N} L(acc_i, \gamma)$ FOR $m = 1$ to M FOR $i = 1, .., N$ STATE compute $r_{im} = -\Big[\frac{\partial Lacc_i meta(f_i)}{\partial meta(f_(x_i)}\Big]_{f=f_{m-1}}$ ENDFOR STATE fit a regression tree to the target $r_i m$ FOR $j = 1, 2, ...m$ STATE compute $\gamma_{jm} = arg\min_\gamma \sum_{x_j \in R_{jm}} (acc_i meta_{m-1}(f_{m-1}(x_i) + \gamma)$ ENDFOR STATE update $meta_m(x) = meta_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$ ENDFOR STATE output $\tilde{meta}(x) = meta_M(x)$

In Fig 4.9, relative importances of the most important features are presented. The most important feature is assumed as 100% and all other features importance factors are scaled to $[0, 100]$. The value of runtime for the previously completed job from the same user has the most influence on the predictability of the runtime for the current job. We see that the requested runtime by users and the runtime for the job before the last job from same user are the second and third important features. The features names are presented in Table 4.1.

Correlation of estimated accuracy values with the actual accuracy values have been used to measure goodness of fit for meta learning approaches in the literature (20). In Table 4.2, the correlation of actual runtime accuracies and estimated accuracies are compared with CNK (20) and decision tree regressor. We observe that gradient boosting tree meta learning approach is the most appropriate for prediction reliability

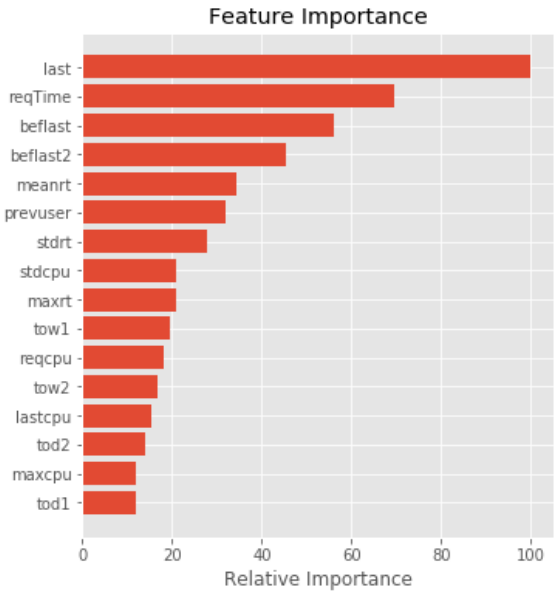

Figure 4.9: Feature importances for meta learning model are shown.

	gradient boosting tree	decision tree	CNK
HPC2N	0.84	0.77	0.44
SDSC	0.81	0.75	0.53
ANL	0.79	0.68	0.47
LLNL	0.78	0.66	0.50

Table 4.2: Correlation of estimated accuracies with actual accuracies for gradient boosting tree is compared with decision tree and CNK.

estimation of job runtime values.

4.5 Evaluating the Performance of our Proposed Hybrid-Scheduling Platform

We use four well-known real-world production traces from (wor) to evaluate our algorithms. HPC2N is trace log containing three and a half years worth of accounting records from the High-Performance Computing Center North (HPC2N) in Sweden. LLNL ATLAS is trace log from ATLAS cluster in Lawrence Livermore National Lab and ANL Intrepid is from Intrepid cluster in Argonne National Lab. SDSC trace is from the SDSC Blue Horizon in San Diego Super Computer. In order to avoid overfitting our machine learning models, we perform our statistical analysis on a separate trace log (HPC2N) and test our machine learning models on three other trace logs.

4.5.1 Event-Driven Simulation

We simulate the scheduling of traces using the open source event-driven simulation package, Alea2 (83). Alea2 simulator is based on the GridSim simulation toolkit (24). Alea2 extends Gridsim to provide a simulation environment that supports simulation of varying job scheduling problems (83). Alea2 uses a centralized job scheduler which uses scheduling techniques for schedule generation. For priority-based scheduling algorithms, Alea2 central scheduler, jobs are submitted according to their submission time in the trace. An ordered queue of jobs is maintained and is updated at each event of job submission or job termination. Once started, jobs run to completion, implying that the central scheduler is not allowed to preempt or migrate the tasks. We have added an online regression method as a machine learning based prediction module and gradient boosting tree as job runtime prediction accuracy estimation to the Alea2 package. At the time of each job submission, the prediction module will calculate the prediction of runtime and returns the runtime value to the central scheduler.

4.5.2 Comparison with Existing Scheduling Approaches

The aim of this section is to compare the performance and efficiency of our proposed hybrid scheduling algorithm with common scheduling algorithms in a real-world scenario. For this purpose, we implemented hybrid scheduling platform as described in Section 4.4. To implement the platform, existing prediction module in Alea2 was extended with implementing online normalized regression as described in (55) as well as a module for performing reliability estimation with gradient boosting tree regression. A thin module of Hybridization Parameter Calculator was also added to ALEA 2 repository. In scheduling module, hybrid scheduler as well as FCFS backfilling with SVF and online-SVF was added.

In the performance and efficiency comparison experiments, hybrid scheduling algorithm was compared with two backfilling approaches as well as two plan-based scheduling algorithms on production traces. *FCFS-BF1* denotes backfilling with SJF as used by (142) and (55). *FCFS-BF2* is similar to *FCFS-BF1* but uses SVF (Smallest Volume First) rule to choose jobs to backfill. *Plan-based1* is the dynamic version of SJF as described in Section 4.2.1. *Plan-based2* is dynamic version of SVF (Im et al.).

Our Hybrid-platform uses Online-SVF as the plan-based scheduling algorithm and FCFS-Backfilling as the backfilling algorithm. SVF is chosen for list scheduling and for backfilling algorithm, we chose FCFS-SVF as discussed in Section 4.2.1. Waiting time and bounded slowdown are among the most used criteria for evaluating the performance of the scheduling algorithm and utilization is a criterion to evaluate the resource usage efficiency of the scheduling algorithm. The criteria are listed below.

Bounded slowdown: The slowdown of a job is the ratio of job response time to its actual time. As this formula emphasizes on the short jobs with runtime near zero, (142) has proposed bounded slowdown. Bounded slowdown substitutes the runtime in the dividend by the maximum of a constant value, τ and the actual runtime.

$$blsd = \max\left(\frac{wait_j + p_j}{\max(p_j, \tau)}, 1\right) \tag{4.6}$$

Average wait time: Average wait time measures the average of time between job submission and the job start time on the system overall jobs.

Utilization: The ratio of total node hours used by the scheduling algorithm to the total nod hours elapsed from the time the first job was submitted to the system. We used Hybrid-platform to schedule jobs of four different traces from parallel workload dataset and compared the wait times with FCFS and Backfilling algorithms SJF-BF, and SVF-BF as well as plan-based scheduling algorithms, online-SJF and online-SVF. We can see in Fig 4.10 that HS has about 20% lower wait time on average. In Fig 4.11 and Fig 4.12, bounded slowdown and utilization improvement of Hybrid-platform is compared with backfilling and plan-based approaches. We can see that Hybrid-platform's bounded slowdown is more than 50% lower than backfilling and online list scheduling methods. The utilization is reported as an improvement over simple SJF scheduling. We can see in Fig 4.12 that utilization improvement is also significantly higher than the backfilling approaches.

4.5.3 The effect of Clairvoyance on Hybrid Scheduler

In Table 4.3, we compare the bounded slowdown of Hybrid-platform with best performing backfilling and plan-based algorithms both in clairvoyant case where accurate runtime values are available at the time of job submission as well as non-clairvoyant case where runtimes are predicted with system generated approaches. Plan-based

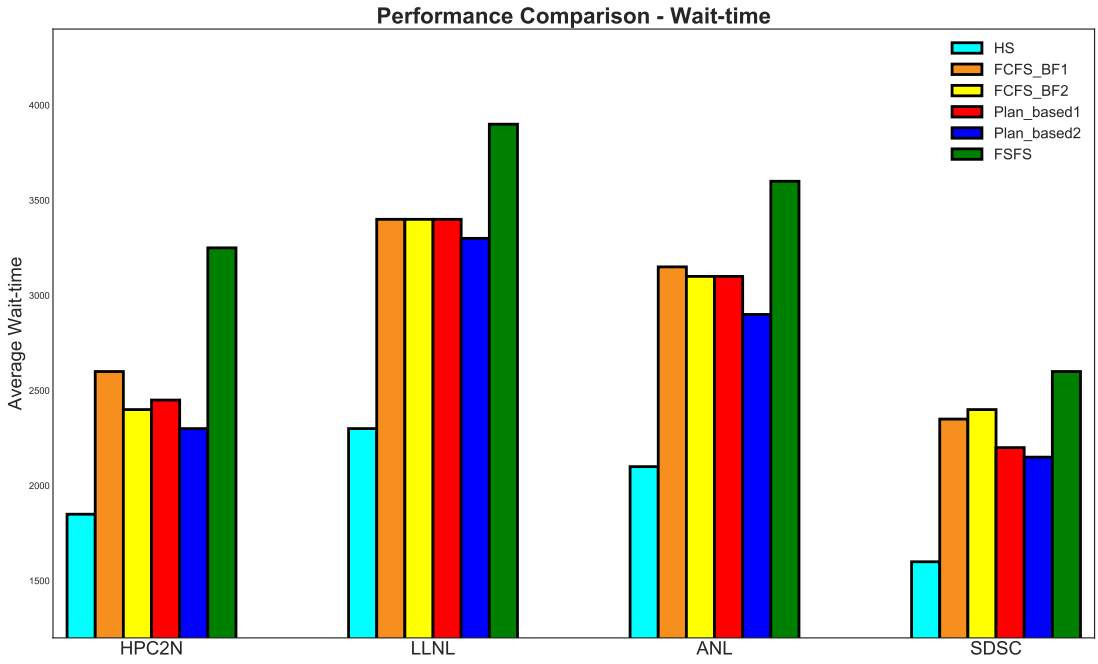

Figure 4.10: Wait time values of HS is compared with FCFS, SVF-BF, SJF-BF, SVF and SJF.

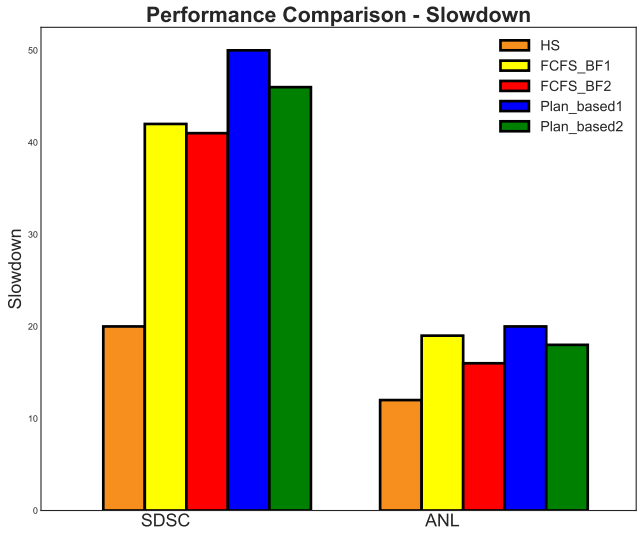

Figure 4.11: Bounded slowdown values of HS is compared with SVF-BF, SJF-BF, SVF and SJF.

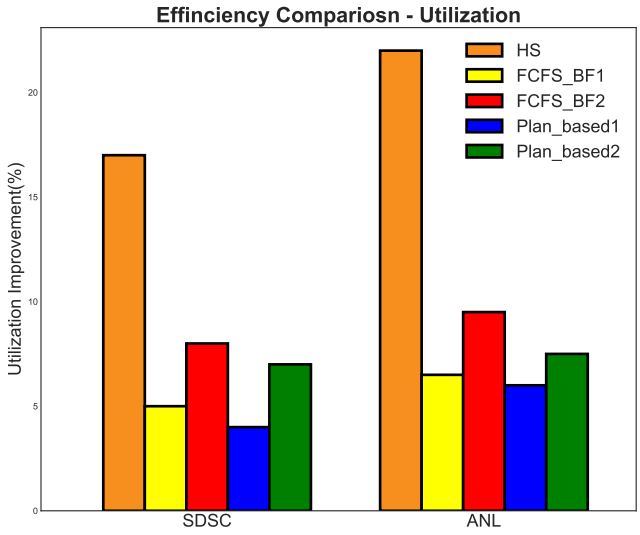

Figure 4.12: Utilization percentage of HS is compared with SVF-BF, SJF-BF, SVF and SJF.

	Clairvoyant		Nonclairvoyant		
	Backfilling	Plan-Based	Backfilling	Plan-based	Hybrid-platform
SDSC	40	20	50	45	25
ANL	24	10	24	22	12

Table 4.3: Bounded-slowdown comparison with Clairvoyant problem setting.

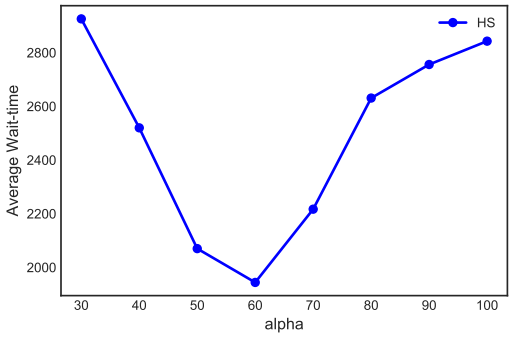

Figure 4.13: Comparing average wait time with various initial alpha (α_0)

scheduling performs best in the clairvoyant setting when accurate job runtimes values are available. However, in the more realistic situations, we can see that Hybrid-platform is performing very close to the ideal case.

4.5.4 Parameter Selection for Hybrid Scheduling

In order to test our heuristic approach, we studied the changes in waiting time with different initialization of α. Although the average wait time was not sensitive to initial parameter, α_0, the initialization of $\alpha_0 = 0.6$ resulted in better performance.

4.6 Summary

The goal of this chapter was to design a scheduling platform to handle inaccuracies in workload runtimes. We designed a novel scheduling platform that hybridizes two popular classes of scheduling algorithms namely, backfilling and plan-based scheduling. Our design was motivated by different sensitivity of these two classes of scheduling algorithms to runtime prediction accuracy. Our platform is designed based on a deep understanding of the characteristics of plan-based and backfilling scheduling algorithms. To avoid resource fragmentation, our proposed platform is adaptive and dynamically changes the portion of computation resources that plan-based scheduler uses based on resource requirement ratio of recently submitted jobs with reliable run-

time prediction. Our extensive trace-based experiments show significant improvement in the performance and utilization of HPC workload traces.

While our present work serves as a demonstration of using prediction reliability to improve performance and utilization of HPC scheduling platforms, there are several directions for taking our idea further. A natural direction is to implement the proposed platform in scheduling packages such as Cobalt (38). Another direction is to extend the proposed approaches to consider prediction reliability for resource consumption of data center workloads.

Chapter 5

Predicting Runtimes with Hierarchical Kalman Filters

We propose adaptive online application runtime prediction methods to improve application latency in clouds. Scheduling algorithms are highly sensitive to the value of application runtime. It is well known that accurate prediction of job runtimes improves scheduling performance. Previous studies assume Gaussian distribution for runtime values and build their predictive models based on this assumption. We show how parallel workload runtimes follow a multi-modal distribution and propose using Gaussian Mixture Models to achieve more accurate prediction. We model the joint distribution of runtimes and collective features available for corresponding jobs. In order to achieve the appropriate parameterization of the Gaussian Mixture Models, we use deep neural networks to find the best parametrization of the mixture model. Our experiments show that Deep Mixture Density Network (DMDN) is capable of more accurate prediction of runtime values given the appropriate features from the newly submitted job.Our machine learning prediction results on HPC application traces show that our adaptive online prediction models predict the runtimes 33% to 80% more accurately than existing prediction approaches. Besides, our extensive trace-based scheduling simulations show that our predicted runtimes improve the performance (wait time) by 25%.

5.1 Introduction

With the increasing availability of cloud computing services, public and private organizations are increasing their use of cloud for High-Performance Computing (HPC) applications to avoid up-front expenses of deploying local computation clusters. Accurate prediction of application runtimes improves the performance of HPC scheduling. More specifically, helps to increase the cloud vendor revenues substantially. To optimize cloud vendor revenues on the computation of HPC applications, it is necessary to apply predictive models that are online, fast, adaptive and accurate. Additionally, it is required that these prediction models work with minimal training data.

We consider the problem of predicting HPC workload runtimes to improve scheduling performance and efficiency. We studied HPC application traces intensively and used our machine learning insight to design a predictive model to estimate HPC application runtimes. Inherent variance and dynamic nature of applications make the accurate runtime prediction a challenging problem. With the fast changing dynamics of applications, training a static model based on the pool of available completed applications and applying it to predict new applications results in inaccurate prediction. The scarcity of training data is another challenge. Most of the existing works on predicting cluster application runtimes predict the runtimes with profiling or analyzing the content of the submitted applications. These approaches are not appropriate for public cloud due to low computation overhead requirement and privacy considerations.

Recently, with the exponential increase in the volume of streaming data, there has been a boost in designing online machine learning methods for time series data. These approaches can be incorporated into cloud resource management systems to predict resource usage and design resource allocation and scheduling algorithms that maximize the cloud vendor revenue. In this paper, we focus on predicting the runtime of HPC applications submitted to the cloud. We propose tailored machine learning approaches specific to the dynamicity and heterogeneity of HPC applications submitted to the cloud to improve the accuracy of runtime prediction for these applications.

To address these requirements, we propose multi-model generative online prediction approaches to predict execution runtimes of applications in the cloud. The choice of generative approaches is because the generative models are well known for handling prediction of data with non-static underlying distribution. Our generative models have the capacity of incorporating time-varying parameters instead of static parameters. This will gives the on-the-fly change of model and makes the model adaptive. The specific generative models that we consider, known as State Space Models are known for handling irregularly spaced data which matches application runtime time series that we are considering. Additionally, as applications submitted to the cloud follow multiple patterns, model selection is an indispensable part of the prediction approach. Traditionally model selection is applied based on the decision of domain experts. However, this is not possible for real-time prediction of applications submitted to the cloud. With designing multiple model approaches, we facilitate automated model selection on-the-fly.

After reviewing related work on runtime prediction and its impact on HPC scheduling in Section 5.2, we present our adaptive online prediction model in Section 5.3. We evaluate our prediction models in Section 5.4 and Section 5.5. in Section 5.4, we compare the prediction accuracy of our proposed methods on HPC traces. Additionally, we incorporate our prediction methods into open source Alea2 simulation package and evaluate the effectiveness of our more accurate prediction runtimes by evaluating the performance improvement for popular scheduling algorithms.

5.1.1 Main Contributions of the Chapter

First, we propose two novel generative multi-model machine learning approaches for adaptive online runtime prediction of HPC applications in the cloud. These multi-model approaches are fairly generic and are expected to be used with wide range of applications submitted to the cloud. The hybrid models are proposed for Gaussian linear state space models but are suitable for more general state space models. We incorporated adaptive prediction by using dynamic models rather than existing static models to decrease prediction bias for high variance nature of cloud application data. Second, since designing hybrid dynamic systems for application runtime prediction is a nontrivial effort, we dedicate main part of the paper to describe the details of the formulation of our two recursive filtering approaches, exponential smoothing, and recursive linear regression and explain the design of multi-model approaches composed of these building blocks. Third, we conduct simulation experiments on real-world HPC traces to evaluate the effectiveness of our approaches and their advantages over existing prediction approaches. Our results show that our adaptive online prediction models outperform the existing methods by a considerable margin. We also demonstrate the effectiveness of our more accurate prediction models on efficiency and performance of common scheduling algorithms for HPC applications. Our trace-based simulations show that using our predictions reduces the average waiting time and response time considerably.

5.2 Related Work

The necessity of runtime prediction for parallel applications has been highlighted since last years of 20th century(58). Different approaches have been proposed to predict HPC application runtimes with different machine learning methodology as well as prediction features inputs (142; 139).

Some other researchers have considered additional features for each job in trace and perform prediction based on the similarity between these features of the jobs (102; 125). Some more recent works focus on the specific family of scientific workflows and use machine learning for predicting runtime and resource usage for these applications (100). Several works have proposed interrogating the codes to extract features for runtime prediction. This is also not practical in many cases due to privacy considerations. All these approaches are static, meaning that they train a static model based on the available applications and use the model for prediction the runtimes of new applications. Applying such a static model for a dynamic environment like cloud leads to inaccurate predictions.

To configure the model based on the individual user and use the recently completed jobs in the same trace to strengthen the predicting power of our model, several online methods have been proposed (133; 139). Several time series based methodologies are proposed to use data from previous jobs to predict the runtime for newly submitted applications. They are mainly exponential smoothing and moving-average

methodologies that predict future values based on the recent runtime values. As noted by (133), these methods are not accurate and will not improve the scheduling performance and utilization significantly. In fact, the scarcity of relevant training data makes the model building cumbersome. The majority of online time series based prediction methods use simple forecasting rules including mean, moving average and exponential smoothing. Sonmez et al. partition jobs into jobs submitted by the same user or jobs running at the same site and applies simple time series methods to predict the subsequent runtime of jobs based on the recent history. They consider mean, running mean of the last two jobs as the prediction method. Although these methods are easy to implement and do not need a large training pool, they are not very accurate.

Most similar work to ours is (55), where an online discriminative approach is proposed to predict runtimes for HPC applications in a parallel computing platform. In their work, they consider historical data including few recent application runtimes as input features to an online polynomial regression. They consider several settings for their model and use the available traces for manual model selection. Our proposed online generative prediction approach considers both historical runtime data as well as trace based features. The multi-model design allows our approach to perform on the fly model selection and achieve faster convergence to accurate model. Furthermore, our model is more robust to drifting.

In this work, we solely focus on appropriate modeling and adaptive prediction of HPC application runtimes. Several works have addressed the characterization of applications in cloud (118; 103; 39). Their studies shed more light on heterogeneity and dynamicity of applications in clouds. Several prediction-based scheduling approaches including (19) and (37) perform application profiling to gather information about the current application. However, these approaches are not practical for large-scale deployment. There are several works including (111) on interference detection of applications which is not applicable to our problem setting (single application per virtual machine).

5.3 Adaptive Online Machine Learning for Application Runtime Prediction

5.3.1 Overview

Our goal is to predict the runtime for each HPC application right after it is submitted to the cloud. Based on recent advancements in cloud virtualization technologies and Software Defined Networks(SDN) (91; 156), we make three important assumptions in this paper:

- Granularity of application: we consider the HPC applications with the granularity of applications and we do not look into task level specifications.

- Application level virtualization: a new virtual machine (VM) is deployed for each application and terminated as the related application completes execution.
- Non-preemption: a running application/VM will not be interrupted or migrated until the completion of execution.

In this work, we are not concerned with the details of tasks in each HPC application and focus on these applications as a single unit. As our prediction methods are fairly generic they can be extended to a finer granularity of tasks as future works. Note that light-weight virtualization services such as containers can facilitate the allocation of each application on a single virtual machine. So we assume a *VM* starts as soon as an application is submitted to the system and terminates when the application completes execution. As data extensive nature of HPC applications makes preemption costly and impose overhead on the computation pipeline, we assume applications to be non-preemptive.

We consider the problem of application runtime prediction for a newly submitted HPC application request. To perform the prediction we have access to available information from application requests as well as system features and history of previous application runtimes. The training data is gradually accumulated and we need to perform a prediction for each new feature vector right away with the available data. For each application A_i, the feature vectors, Z_is, enter to the system at time t_i. Its actual runtime, y_i is only known after the A_i's completion time t'_i ($t'_i > t_i$).One naive approach is to apply blind predictors for the first applications and gather the completion times to build a model for prediction runtimes of future applications. The offline model will be retrained after several new application features and actual runtimes are accumulated. However, as it seems, this approach is not accurate. Increasing the frequency of retraining may increase the accuracy but makes the model building overhead intractable. The appropriate approach for performing prediction is performing online learning. The general scenario is that the prediction is performed for a new observation using a prediction model. After the actual application runtimes become available, the prediction error, $v_i = y_i - \hat{y}_i$ is used to tweak the model for more accurate prediction of future application runtimes. The online prediction approaches can be classified into two general groups of discriminative and generative approaches. In discriminative approaches, the new model parameters are updated with minimizing the loss function. However, discriminative models are known to be inaccurate in case of limited training data. They are also known to be ineffective when the underlying distribution of data is not stationary and evolves over time. As submitted applications to the cloud are known for dynamicity and constantly changing, we propose generative online learning approaches. In generative approaches, an initial underlying distribution is assumed for the application runtime data and will be updated with each new feature vector and actual runtime. Using generative approaches, we can better detect distribution changes in the joint distribution of runtimes and features data. Generative models let us use Bayes rule to perform automatic model selection as we will describe in our first proposed approach. We can also consider multiple

Table 5.1: Features extracted from SWF files of HPC application traces for each user

Feature	Meaning
$\tilde{y_j}$	User estimated runtime for jth application application requested by the
d_j	The resource (CPU) request for jth application application requested by th
TD	Time of the day the application is submitted
TW	Time of the week the application is submitted
$Free_{capcity}$	Available free resource at the time of submission normalized by total resources for H

models for each trace and use observed features to perform model selection for each application in the trace. Additionally, the generative approaches are known to perform better in the case of missing data which is common in our problem.

Problem Statement We describe the applications in terms of historical data in time series format. time series are sets of runtimes $y_1, ..., y_T$ and features $Z_1, Z_2, ..., Z_T$ ordered in time. Each application is indexed by the time it is submitted to the system. Different features including runtime and resource usage can be expressed as time series. The applications can be described with the following triplets.

$$(t_i, y_i, Z_i) \mid 0 \leq i \leq n$$
$$t_i \in \mathcal{R} : t_i \leq t_{i+1}$$

The features (Z_i) are described in Table 5.1. Part of features are extracted from application description including required resources. Some system features and environmental features including number of jobs currently running on the system and time of the day and day of the week are also considered.

5.3.2 Prediction Methodology

The prediction is achieved via modeling the applications runtimes time series as a state space model. This choice is motivated by the dynamic nature of state space models that facilitates an adaptive prediction of application runtimes. State space models can be considered as generative machine learning approaches where the set of observations are generated from a set of hidden states that evolve over time. In our study, the unobserved series that we call latent properties are application runtimes which are unknown before the application completes execution in the system. Regression coefficients (Z_t) are the properties that are known at the time of application submission, including submission time, user estimate of runtime, user estimate of CPU and requested memory. We assume that the over time evolution of the application runtimes denoted by $y_1, ..., y_t$ are associated with a series of observed features $Z_1, ..., Z_t$. We consider runtime values y_i as regressions of observed features Z_is plus some error ϵ as in Equation 5.1. This is exactly the recursive least square. However

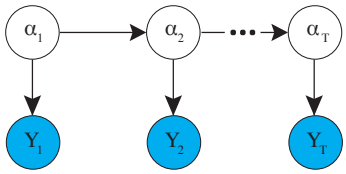

Figure 5.1: The Kalman Filter is a Hidden Markov Model with continuous latent variables. Y_is are observations produced by X_is. X_i form a random walk.

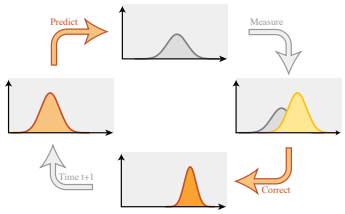

Figure 5.2: Kalman Filter starts with an initial distribution, after each observation, the assumption is filtered to a more accurate distribution.

we consider a more general formulation where the regression coefficient, α_i is not constant and changes according to a Markov chain as in Equation 5.2.

$$y_t = Z_t\alpha_t + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \mathcal{H}_t) \tag{5.1}$$

$$\begin{aligned} \alpha_{t+1} = T\alpha_t + \eta_t, \quad \eta_t \sim \mathcal{N}(0, \mathcal{Q}_t) \\ \alpha_1 \sim \mathcal{N}(a_1, p_1). \end{aligned} \tag{5.2}$$

Z_t, R_t, H_t and Q_t are initialized based on domain knowledge for the specific model. Note that Equation 5.1 along with Equation 5.2 represents general linear Gaussian state space model . We consider two different settings for the parameters Z_t and T and derive exponential smoothing and adaptive linear regression from this linear dynamic system. The system starts with initial distribution for the hidden value α. As the system gets new observation values, the model parameters α_{t+1} and p_{t+1} are updated using a regression lemma for calculating mean and covariance of conditional distribution of $P(\alpha_t|Y_{t-1}, v_t)$ (43). Given runtime values (y_t) and the error in prediction of y_t, denoted as v_t, we can update our model for the prediction of current application runtime. This recursive process is called Kalman filtering and is presented in Equation 5.3.

$$a_{t+1} = Ta_tZ_t'Z_tP_tZ_t' + H_t \tag{5.3}$$

$$P_{t+1} = TP_t(T - TP_tZ') + Q_t \tag{5.4}$$

At time t, the Equation 5.3 calculates the estimated mean of hidden variable α_t, denoted by a_t, and its covariance, denoted by p_t. Having the mean and covariance of

the hidden variable, we can calculate the prediction of our runtime values using the Equation 5.1.

Motivated by the auto-regressive property of traces and the correlation between available features and the history of previous runtimes, we consider two important groups of linear Gaussian state space models. Autoregressive forecasting (exponential smoothing) and recursive least squares. We highlight the fact that both these models can be considered as special cases of Linear Dynamic Systems, known as Kalman Filters (43). Generalizing these two approaches as state space models are beneficial because it facilitates hybridizing these models and also makes our approaches extendable to more complex models in future studies. We focus on linear Gaussian state space models and propose two multi-model prediction approaches based on these models to reach lower bias and higher prediction accuracy.

Exponential Smoothing: Now we will show that Equation 5.2, which is also called the state space model, is in fact a first order auto-regressive model. Considering the feature vectors Z_t and T as unit vector, we derive exponential smoothing prediction recursion which is a common low-overhead approach to forecast future values of runtimes. The following formulation of exponential smoothing can be expressed in recursive formula of Equation 5.6.

$$y_{t+1} = (1-\lambda)\sum_{0}^{\infty}\lambda^j y_{t-1} \quad 0 \leq \lambda \leq 1 \tag{5.5}$$

$\hat{y}_t$ is calculated with the following recursive relation:

$$\hat{y}_{t+1} = (1-\lambda)y_t + \lambda\hat{y}_t \tag{5.6}$$

The exponential smoothing recursion can be derived from Equations 5.1 and 5.2 as follows:

$$y_t = \alpha_t + \epsilon_t\alpha_{t+1} = \alpha_t + \eta_t y_t - y_{t-1} = \epsilon_t - \epsilon_{t-1} + \eta_t \tag{5.7}$$

Setting λ appropriately will yield the Equation 5.6.

Adaptive Linear Regression: To gain more predictive power, we would like to use the information available at the submission time of the applications. A simple regression model for runtimes is

$$y_t = Z_t\alpha + \varepsilon_t \quad \varepsilon_t, \sim N(0, H_t)$$

which is actually in the form of Gaussian linear dynamic system. where Z_t is $1 \times k$ regressor vector and α is the regression coefficient. We can see that the Gaussian linear dynamic system derived from Equations 5.1 and 5.2 gives the recursive least squares method as developed by Packett (112). To derive the regression with time varying coefficient, we allow the coefficient α to vary according to a random walk, $\alpha_{t+1} = \alpha_t + \eta_t$. The related Gaussian linear dynamic system expression will be

$$\begin{cases} y_t = z_t\alpha_t + \epsilon_t \\ \alpha_{t+1} = \alpha_t + \eta_t \end{cases} \tag{5.8}$$

After deriving the filter equations for mean and variance of α_{t+1}, Equation 5.1 is applied to predict the value of runtime (y_t).

5.3.3 First Proposed Approach: Fixed Multiple Kalman Filter (FMKF)

In our first approach, we perform automated model selection between two different candidate models: exponential smoothing and adaptive linear regression. Our hybrid approach figures out which model is actually producing the runtime values based on the prediction error for previous time steps. It calculates each runtime based on the likelihood of each models being in effect. We follow a Bayesian framework: starting with prior probabilities of each model being correct (the system being in a particular mode), the corresponding posterior probabilities are obtained. We assume the model the system obeys is fixed. The model assumed to be in effect is one of the two candidate models. We explained the Kalman Filter representations of Exponential Smoothing and Adaptive Linear Regression in the previous section. Next, we will present our hybrid prediction model that combines the two models.

Our first hybrid model arbitrates between two predictive models: Exponential Smoothing and Adaptive Linear Regression. We assume that only one of the predictive models is in effect for each user trace. We designed this model based on the assumption that as we get more completed applications from a user, adaptive linear regression gains more power to predict the application runtimes. Probabilities p_1^t and p_2^t determine which of the two Kalman Filters the model is using to generate prediction in step i. The initial probabilities p_1^0 and p_2^0 favor auto-regressive Kalman Filter and both models evolve as more application runtimes are available. The likelihood function of model j at time t is given by

$$\Lambda_j(t) = p[z(t)|Z^{t-1}, M_j] = p[\nu_j(t)] = \mathcal{N}[(\nu_j(t); 0, S_j(t)] \tag{5.9}$$

Where ν_j and S_j are the innovation(error of prediction) and its covariance for model j (11). Using the likelihood for each model we can calculate the probability that model j is effective after tth step by the following expressions.

$$\mu_j(t) = \frac{\Lambda_j(t)\mu_j(t-1)}{\sum_{i=1}^{2}\left[\Lambda_i(t)\mu_i(t-1)\right]} \quad j = 1, 2 \tag{5.10}$$

The state of the system is a Gaussian mixture with two terms

$$p(\alpha(t)|Z^t) = \sum_{i=1}^{2} \mu_i(t)\mathcal{N}[\alpha_i(t); \hat{\alpha}_i(t), P_i(t)] \tag{5.11}$$

Mean and covariance of the state at time step t is calculated using the most recent posterior probability of each model $\mu_j(t-1)$. For derivations of the mean and covariance of the mixture model we refer the reader to the work of Bar-Shalom et al. (11).

5.3.4 Second Proposed Method: Multi-Layer Kalman Filter (MLKF)

Our second approach is a hierarchical Kalman Filter based on the observation that most runtime traces for users follow a bimodal distribution, as an example the dis-

tribution of application runtimes is demonstrated Fig 5.3. At the first level, the application is classified as *short* or *long*. In the second level, one of the two auto regressive Kalman Filter is applied for predicting runtime. We initialize two auto

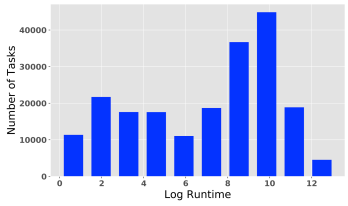

Figure 5.3: Runtimes of the all jobs are shown.

regressive Kalman Filters that evolve in parallel over time. At each time step, one of the models is creating runtime values. To predict the runtime for a new application we should determine which model to use for prediction. This prediction takes place in the first layer of our prediction model, where a classifier Kalman Filter is used to determine which of the two models is generating the runtime values. This first layer, is actually an adaptive online classifier that uses available features at the time of job submission to decide which model should be used to predict the runtime for the submitted application. This is shown in Fig 5.4. To design the online adaptive classifier, we use a modified version of adaptive linear regression Kalman Filter. The modification is related to mapping the Kalman Filter output to a class label. Our online classifier, classifies the incoming feature vector into one of the two classes: *long* and *short*. The online adaptive classifier generates a binary value by thresholding the result of an adaptive linear regression Kalman Filter.

$$C(y_i) = \begin{cases} 1 & if \quad y_i \geq T \\ 0 & Otherwise1. \end{cases} \tag{5.12}$$

The adaptive online classifier thresholds the result of the adaptive online classifier as in Equation 5.8 as in equation 5.12 determine the class that the input observation belongs to. In the second level, based on the class the application fall into, the related autoregressive Kalman Filter is applied to predict the runtime. Consider an application a is submitted to our Multi-Level Kalman Filter Model (MLKF). At the

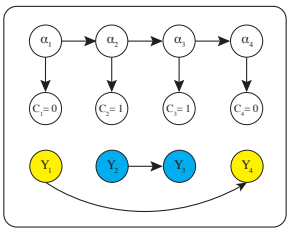

Figure 5.4: One of the two auto regressive Kalman Filters $AR1$ and $AR2$ are chosen based on the classification of features by Adaptive Online Classifier.

first level, an adaptive logistic regression will determine if the incoming workload belongs to the *long* class or *short* class. At the second level, the appropriate Kalman Filter is applied to predict the runtime of a.

5.4 Experimental Evaluation of the Prediction Methods

We implemented our novel prediction models as a Java module for ALEA2 simulation package. We have shared the updated package in Github. In this section we report our trace based experiments to showcase the effectiveness of our proposed prediction models.

Trace Data: We use four widely used real world production traces from (wor) to evaluate our algorithms. HPC2N is trace log containing three and a half years worth of accounting records from the High-Performance Computing Center North (HPC2N) in Sweden. LLNL ATLAS is trace log from ATLAS cluster in Lawrence Livermore National Lab and ANL Intrepid is from Intrepid cluster in Argonne National Lab. SDSC trace is from the SDSC Blue Horizon in San Diego Super Computer. In order to avoid overfitting our machine learning models, we perform our statistical analysis on a seperate trace log (HPC2N) and test our machine learning models on three other trace logs.

5.4.1 Prediction Accuracy Evaluation

We compare the accuracy of our proposed online generative methods with most common prediction method for parallel workload scheduling. We will demonstrate how our accurate predictions improve the performance of scheduling algorithms in section 5.5. We follow the commonly used measure of prediction accuracy in HPC scheduling literature(142; 133).

$$accuracy = \begin{cases} 1 & if \quad \hat{y} = y \\ \frac{\hat{y}}{y} & if \quad \hat{y} < y \\ \frac{y}{\hat{y}} & if \quad y < \hat{y} \end{cases} \tag{5.13}$$

We compared the accuracy of our proposed adaptive models with exponential smoothing (ES) and moving average($MA2$) as the most widely used methods of online prediction for job runtime. MA calculates the current runtime as the average of the runtime of the last two completed jobs. ES Calculates a weighted average of previous values. We see that both FMKF and MLKF improve accuracy substantially over the common widely used online runtime prediction methods. For all user traces FMKF and MLKF perform substantially better than common approaches for predicting runtimes.

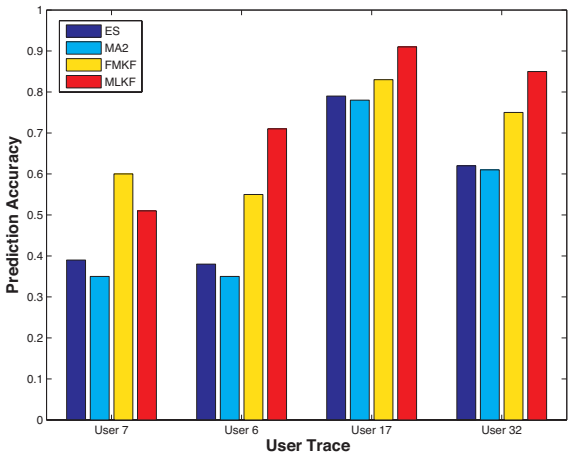

Figure 5.5: Prediction Accuracy of several traces in HPC2N trace

5.5 The impact of More Accurate Predictions on Scheduling Performance

Recently, consolidation of cloud applications have been proposed to improve resource efficiency and power usage in data centers(35; 134). These consolidation methods consider a limited set of resources for a set of jobs. This setting is similar to that of HPC clusters. More recently, Software Defined Networks provide the abstraction for assigning a block of virtual resources to a set of specific type of jobs (86). Scheduling algorithms that optimize performance are highly dependant on the accurate prediction of application runtimes. In this section, we investigate the performance of wait time and response time in HPC traces. We consider an HPC cluster managed by a centralized scheduler that has complete control over all jobs and resources in the system. The events in the system include application submission, application start, normal application end and application termination if the job exceeds its expected runtime.

5.5.1 Event Driven Simulation

We simulate the scheduling of traces using open source event-driven simulation package, Alea2(83). Alea2 Simulator is based on the GridSim simulation toolkit which was extended to provide a simulation environment that supports simulation of varying job scheduling problems. Alea2 uses a centralized job scheduler which uses (advanced) scheduling techniques for schedule generation. For priority based scheduling algorithms, Alea2 central scheduler, jobs are submitted according to their submission time in trace. An ordered queue of jobs is maintained and is updated at each event of job submission or job termination. Once started, jobs run to completion, implying that the central scheduler is not allowed to preempt or migrate the tasks. We have

added our implementation of the predictive methods as a machine learning based prediction module to the Alea2 package. At the time of each job submission, the prediction module will calculate the prediction of runtime and returns the runtime value to the central scheduler.

5.5.2 Scheduling Algorithms

The scheduling scenario we consider here is as follows: there are n applications: $A_1, A_2, ..., A_n$ each A_i has a resource demand d_i and runtime p_i. Applications are submitted to the system at submission times $t_1, t_2, ..., t_n$. Each A_i will have response time r_i and wait time w_i. The former is the time it takes for the application form submission to completion, the latter denotes the time that takes for the job to start execution after submission to the system. We assume that the total resource available is R. For simplicity, we only assume one type of resources. A scheduling algorithm assigns a time s_i to each application. We are interested in scheduling that optimizes the sum of response times $\sum_i r_i$. The most widely used for HPC applications are priority based algorithms. In priority based methods, an ordered queue of jobs is maintained and updated at each event of new job submission or job termination. Priority based scheduling algorithms suffer from fragmentation of resources. This leads to having many idle processors which imply lower performance and resource utilization. Backfilling methods are proposed to improve priority based scheduling policies by allowing waiting jobs in the queue to bypass earlier jobs as they can fit into the idle resources. The most popular backfilling algorithm in HPC clusters is Easy backfilling. It allows later jobs in the queue to move forward without slowing the first job in the queue. Easy is widely used because it is simple and has higher performance than plain First Come First Served algorithm. In (142), authors proposed Short Job First Backfilled First (SJF-BF) to leverage runtime predictions. However, in (133), the authors claimed that runtime predictions are inaccurate and do not improve performance and efficiency. Priority-based algorithms are the most widely used scheduling algorithms for HPC clusters due to their low overhead. The most widely used scheduling algorithm is FCFS-Easy. We will showcase the effect of our online machine learning prediction methods with the SJF-BF algorithm. SJF-BF (142) is similar to Easy, except that for the backfilling, the smallest job is selected to bypass the earlier jobs in the queue. To avoid premature killing of jobs according to short predictions(142), our simulator uses the user estimation when our predicted finishing time is reached and the application is still running.

5.5.3 Results

One of the most important metrics to evaluate performance in a dynamic system is the measure of wait time. Wait time is the time that the job needs to wait until it starts runt time. We have compared the wait time of our two implemented exact plan based scheduling algorithms with EASY backfilling on FCFS for three different production traces. Another measurement of performance is the measure of response

time. Response time measures the time it takes from the submission of the job till the job completion time. Both FMKF and MLKF prediction methods improved the wait time and runtime of user traces significantly. To avoid confusion we only plotted the result of the best method for each trace. We can see that although user estimates do not improve the performance of SJFBF over EASY-BF, our machine learning based predictions reduce wait time and response time significantly. Utilization is another

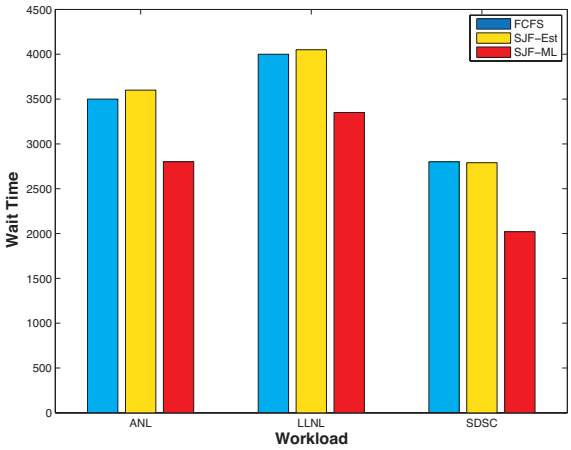

Figure 5.6: Wait time of SJF-BF using ML based predictions.

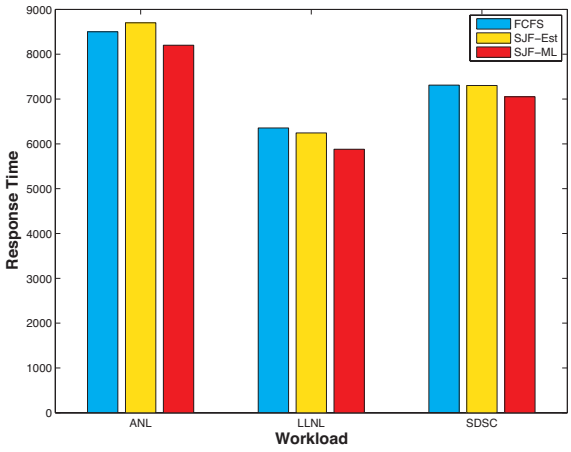

Figure 5.7: Response time of SJF-BF using ML based predictions.

important measure to reduce the OPEX cost of cluster owners. By integrating a fairly accurate prediction system, we were able to improve both of these factors . Table 5.2 shows the improvement of our methods over the commonly used method.

5.6 Summary

The goal of this paper was to explore the suitability of generative online learning methods as adaptive approaches to predict runtime of HPC applications in clouds.

Table 5.2: Comparison of average cluster utilization

Workload	FCFS	SJF-ES	SJF-FMKF	SJF-MLKF
ANL	0.63	0.63	0.64	0.64
LLNL	0.74	0.73	0.75	0.76
SDSC	0.78	0.78	0.81	0.81

We started by generalizing exponential smoothing and recursive least squares methods in form of Kalman Filaters. We then designed two hybrid generative approaches to achieve more accurate prediction for HPC application runtimes in the cloud. In our hybrid approaches, automated model selection and model switching decreased the model bias and facilitated more accurate prediction of runtimes for the HPC application submitted to cloud.

We compared our prediction approaches with widely used approaches on available public HPC traces. Our adaptive prediction models improved over the existing prediction approaches by 33% inaccuracy. Besides, our extensive trace-based scheduling simulations showed that our predicted runtimes improved the performance (wait time) 25%. Our trace-based simulations showed that using the more accurate prediction methods will improve the performance of scheduling algorithms. Our multi-model generative approaches can be used for more general problems of resource usage prediction and improve the performance and efficiency.

Chapter 6

Predicting Runtime using Deep Mixture Density Networks

We propose Deep Mixture Density Networks to prediction HPC application runtimes. Our more accurate predictions improve application latency in HPC clusters. Scheduling algorithms are highly sensitive to the value of application runtime. It is well known that accurate prediction of job runtimes improves scheduling performance. Previous studies assume Gaussian distribution for runtime values and build their predictive models based on this assumption. We show how parallel workload runtimes follow a multi-modal distribution and propose using Gaussian Mixture Models to achieve more accurate prediction. We model the joint distribution of runtimes and collective features available for corresponding jobs. In order to achieve the appropriate parameterization of the Gaussian Mixture Models, we use deep neural networks to find the best parametrization of the mixture model. Our experiments show that Deep Mixture Density Network (DMDN) is capable of more accurate prediction of runtime values given the appropriate features from the newly submitted job.Our machine learning prediction results on HPC application traces show that our adaptive online prediction models predict the runtimes 33% to 80% more accurately than existing prediction approaches. Besides, our extensive trace-based scheduling simulations show that our predicted runtimes improve the performance (wait time) by 25%.

6.1 Introduction

Recently, machine learning approaches have been used for performance prediction for distributed systems. A group of these approaches predicts the performance metric values directly from data available about each application. Another group, first predict performance-related features, including runtime and resource usage. These features are, in fact, the requirement of the application to reach desirable performance. They use these predictions to perform simulations and calculate the values of performance metrics. Performance prediction helps to benchmark different scheduling approaches. In other hands, the need for workload characterization and prediction arise from the

insufficient information at the time of submission of applications. Many scheduling and resource management algorithms require an accurate value of runtime and resource usage to reach to desired performance and efficiency results. As these values are not available, machine learning can be used to provide estimations.

We consider the problem of predicting HPC workload runtimes to improve scheduling performance and efficiency. We studied HPC application traces intensively and used our machine learning insight to design a predictive model to estimate HPC application runtimes. Inherent variance and dynamic nature of applications make the accurate runtime prediction a challenging problem. With the fast changing dynamics of applications, training a static model based on the pool of available completed applications and applying it to predict new applications results in inaccurate prediction. The scarcity of training data is another challenge. Most of the existing works on predicting cluster application runtimes predict the runtimes with profiling or analyzing the content of the submitted applications. These approaches are not appropriate for public cloud due to low computation overhead requirement and privacy considerations.

Our extensive study and analysis of HPC applications traces showed that the target values follow multi-model distribution. More specifically, they can be described with Gaussian Mixture ModelsGMM) effectively. GMMs are good descriptors when a single X can have any of the multiple target values based on the specific Gaussian from the mixture.

Considering the gaussian mixture model for the data, we propose Deep Mixture Density Networks approaches to predict execution runtimes of applications in the cloud. The choice of Deep Mixture Density Networks is because the generative models are well known for handling prediction of data with non-Gaussian distribution. Our mixture models have the capacity of capturing multi-modality of the job runtime distribution. This will gives the on-the-fly change of model and makes the model adaptive. The specific generative models that we consider, known as State Space Models are known for handling irregularly spaced data which matches application runtime time series that we are considering. Additionally, as applications submitted to the cloud follow multiple patterns, model selection is an indispensable part of the prediction approach. Traditionally model selection is applied based on the decision of domain experts. However, this is not possible for real-time prediction of applications submitted to the cloud. With designing multiple model approaches, we facilitate automated model selection on-the-fly.

6.1.1 Our Contributions

We propose the usage of Deep Mixture Density Networks for HPC runtime prediction. The choice of Mixture Density Networks (MDN) is based on our workload analysis and our understanding of the distribution of the HPC runtime data.

6.1.2 Chapter Organization

After reviewing related work on runtime prediction and its impact on HPC scheduling in Section 6.2, we present our Mixture Density Network in Section 6.3. We evaluate our prediction models in Section 6.4. in Section 6.4, we compare the prediction accuracy of our proposed methods on HPC traces.

6.2 Related Work

6.2.1 Related Work on HPC Application Runtime Prediction

The necessity of runtime prediction for parallel applications has been highlighted since last years of 20th century(58). Different approaches have been proposed to predict HPC application runtimes with different machine learning methodology as well as prediction features inputs (142; 139).

Some other researchers have considered additional features for each job in trace and perform prediction based on the similarity between these features of the jobs (102; 125). Some more recent works focus on the specific family of scientific workflows and use machine learning for predicting runtime and resource usage for these applications (100). Several works have proposed interrogating the codes to extract features for runtime prediction. This is also not practical in many cases due to privacy considerations. All these approaches are static, meaning that they train a static model based on the available applications and use the model for prediction the runtimes of new applications. Applying such a static model for a dynamic environment like cloud leads to inaccurate predictions.

To configure the model based on the individual user and use the recently completed jobs in the same trace to strengthen the predicting power of our model, several online methods have been proposed (133; 139). Several time series based methodologies are proposed to use data from previous jobs to predict the runtime for newly submitted applications. They are mainly exponential smoothing and moving-average methodologies that predict future values based on the recent runtime values. As noted by (133), these methods are not accurate and will not improve the scheduling performance and utilization significantly. In fact, the scarcity of relevant training data makes the model building cumbersome. The majority of online time series based prediction methods use simple forecasting rules including mean, moving average and exponential smoothing. Sonmez et al. partition jobs into jobs submitted by the same user or jobs running at the same site and applies simple time series methods to predict the subsequent runtime of jobs based on the recent history. They consider mean, running mean of the last two jobs as the prediction method. Although these methods are easy to implement and do not need a large training pool, they are not very accurate.

Most similar work to ours is (55), where an online discriminative approach is proposed to predict runtimes for HPC applications in a parallel computing platform.

In their work, they consider historical data including few recent application runtimes as input features to an online polynomial regression. They consider several settings for their model and use the available traces for manual model selection. Our proposed online generative prediction approach considers both historical runtime data as well as trace based features. The multi-model design allows our approach to perform on the fly model selection and achieve faster convergence to accurate model. Furthermore, our model is more robust to drifting.

6.2.2 Related work on HPC jobs runtime prediction

In this work, we solely focus on appropriate modeling and adaptive prediction of HPC application runtimes. Several works have addressed the characterization of applications in cloud (118; 103; 39). Their studies shed more light on heterogeneity and dynamicity of applications in clouds. Several prediction-based scheduling approaches including (19) and (37) perform application profiling to gather information about the current application. However, these approaches are not practical for large-scale deployment. There are several works including (111) on interference detection of applications which is not applicable to our problem setting (single application per virtual machine).

6.3 Mixture Density Networks for Runtime Predicion

6.3.1 Overview

Our goal is to predict the runtime for each HPC application right after it is submitted to the cloud. Based on recent advancements in cloud virtualization technologies and Software Defined Networks(SDN) (91; 156), we make three important assumptions in this paper:

- Granularity of application: we consider the HPC applications with the granularity of applications and we do not look into task level specifications.
- Application level virtualization: a new virtual machine (VM) is deployed for each application and terminated as the related application completes execution.
- Non-preemption: a running application/VM will not be interrupted or migrated until the completion of execution.

In this work, we are not concerned with the details of tasks in each HPC application and focus on these applications as a single unit. As our prediction methods are fairly generic they can be extended to a finer granularity of tasks as future works. Note that light-weight virtualization services such as containers can facilitate the allocation of each application on a single virtual machine. So we assume a *VM* starts as soon as an

application is submitted to the system and terminates when the application completes execution. As data extensive nature of HPC applications makes preemption costly and impose overhead on the computation pipeline, we assume applications to be non-preemptive.

We consider the problem of application runtime prediction for a newly submitted HPC application request. To perform the prediction we have access to available information from application requests as well as system features and history of previous application runtimes. The training data is gradually accumulated and we need to perform a prediction for each new feature vector right away with the available data. For each application A_i, the feature vectors, Z_is, enter to the system at time t_i. Its actual runtime, y_i is only known after the A_i's completion time t'_i ($t'_i > t_i$).One naive approach is to apply blind predictors for the first applications and gather the completion times to build a model for prediction runtimes of future applications. The offline model will be retrained after several new application features and actual runtimes are accumulated. However, as it seems, this approach is not accurate. Increasing the frequency of retraining may increase the accuracy but makes the model building overhead intractable. The appropriate approach for performing prediction is performing online learning. The general scenario is that the prediction is performed for a new observation using a prediction model. After the actual application runtimes become available, the prediction error, $v_i = y_i - \hat{y}_i$ is used to tweak the model for more accurate prediction of future application runtimes. The online prediction approaches can be classified into two general groups of discriminative and generative approaches. In discriminative approaches, the new model parameters are updated with minimizing the loss function. However, discriminative models are known to be inaccurate in case of limited training data. They are also known to be ineffective when the underlying distribution of data is not stationary and evolves over time. As submitted applications to the cloud are known for dynamicity and constantly changing, we propose generative online learning approaches. In generative approaches, an initial underlying distribution is assumed for the application runtime data and will be updated with each new feature vector and actual runtime. Using generative approaches, we can better detect distribution changes in the joint distribution of runtimes and features data. Generative models let us use Bayes rule to perform automatic model selection as we will describe in our first proposed approach. We can also consider multiple models for each trace and use observed features to perform model selection for each application in the trace. Additionally, the generative approaches are known to perform better in the case of missing data which is common in our problem.

Features extracted from SWF files of HPC application traces for each user

- $\tilde{y_j}$: User estimated runtime for jth application application requested by the user

- d_j: The resource (CPU) request for jth application application requested by the user

- TD: Time of the day the application is submitted
- TW: Time of the week the application is submitted
- $Free_{capcity}$: Available free resource at the time of submission normalized by total resources for HPC applications

Problem Statement We describe the applications in terms of historical data in time series format. time series are sets of runtimes $y_1, ..., y_T$ and features $Z_1, Z_2, ..., Z_T$ ordered in time. Each application is indexed by the time it is submitted to the system. Different features including runtime and resource usage can be expressed as time series. The applications can be described with the following triplets.

$$(t_i, y_i, Z_i) \mid 0 \leq i \leq n$$
$$t_i \in \mathcal{R} : t_i \leq t_{i+1}$$

The features (Z_i) are described in Table 6.3.1. Part of features are extracted from application description including required resources. Some system features and environmental features including number of jobs currently running on the system and time of the day and day of the week are also considered.

6.3.2 Prediction Methodology

The prediction is achieved via deep mixture density networks (DMDN). Mixture Density Networks are first proposed by Bishop in 1994 (17). MDNs extend the conventional neural networks for approximating target values that do not follow Gaussian distribution. Instead of estimating the average, the network estimates the conditional distribution of the target value. As HPC runtime values follow multi-modal distribution and they show heteroscedasticity, MDNs are appropriate approach for their prediction. Instead of single Gaussian, the general distribution we consider for job runtimes is as follows:

$$p(t \mid x) = \sum_{i=1}^{m} (t \mid x)(t) \tag{6.1}$$

where.

$$\phi(t \mid x) = \frac{1}{(2\pi)^{c/2}\delta_i(x)^c} \exp - \frac{\|t - \mu_i(x)\|^2}{2\sigma_i(x)^2}. \tag{6.2}$$

μ_i represents the center of the i$^{\text{th}}$ Gaussian kernel. A Gaussian mixture models with kernels given by 6.2 can approximate any given density function to arbitrary accuracy provided the mixing coefficients and Gaussian parameters are correctly chosen (101). One important issue is how to determine the parameters of of Gaussian mixture models in 6.1. Following the MDN we consider these parameters to be function of input

data. In order to approximate the function we train a neural network that outputs three sets of parameters to model our mixture model. By choosing appropriate number of mixture models and appropriate architecture of neural network, we are able to approximate conditional distribution of $p(t \mid x)$.

After training the neural network with sufficient input data and corresponding target values, the neural network can be used to approximate target values for input data. Assuming that the component kernels of the mixture model are not strongly overlapping, the target values can be calculated as the center of the highest component calculated as

$$\max_i \{\frac{\alpha_i(x)}{\delta_i(x)}\} \tag{6.3}$$

6.3.3 Architecture of Deep Mixture Network to predict job runtimes

To obtain the parameters for the mixture, a DNN is modified to output multiple parameter vectors. We start off with a single layer DNN and a ReLU activation. Using the hidden layer $h1(x)$, we proceed by computing the parameters of the mixture as follows:

$$h_1(x) = max(W_1^T X + b_1, 0) \tag{6.4}$$

$$\alpha_x = softmax(W_\alpha^T h_1(x) + b_\alpha) \tag{6.5}$$

$$\mu(x) = W_\mu^T h_1(x) + b_\mu \tag{6.6}$$

$$\sigma(x) = W_\sigma^T h_1(x) + b_\sigma \tag{6.7}$$

The mixing coefficient must sum to unity: $\sum \alpha(x) = 1$. Therefore, we are using a softmax function to constrain the output. This step is important, as the mixture of probabilities must integrate to one. The constraints for μ and σ themselves depend on the distribution we are choosing for our model. The only constraint we must enforce for Gaussian is, that the std. deviation is $\sigma(x) > 0$.

6.4 Experimental Evaluation of the Prediction Methods

We implemented our novel prediction models as a Java module for ALEA2 simulation package. We have shared the updated package in Github. In this section we report our trace based experiments to showcase the effectiveness of our proposed prediction models.

Trace Data: We use four widely used real world production traces from (wor) to evaluate our algorithms. HPC2N is trace log containing three and a half years worth of accounting records from the High-Performance Computing Center North (HPC2N) in Sweden. LLNL ATLAS is trace log from ATLAS cluster in Lawrence Livermore National Lab and ANL Intrepid is from Intrepid cluster in Argonne National Lab.

SDSC trace is from the SDSC Blue Horizon in San Diego Super Computer. In order to avoid overfitting our machine learning models, we perform our statistical analysis on a separate trace log (HPC2N) and test our machine learning models on three other trace logs.

6.4.1 Prediction Accuracy Evaluation

We compare the accuracy of our proposed online generative methods with most common prediction method for parallel workload scheduling. We follow the commonly used measure of prediction accuracy in HPC scheduling literature(142; 133).

$$accuracy = \begin{cases} 1 & if \quad \hat{y} = y \\ \frac{\hat{y}}{y} & if \quad \hat{y} < y \\ \frac{y}{\hat{y}} & if \quad y < \hat{y} \end{cases} \tag{6.8}$$

6.5 Summary

The goal of this paper was to explore the suitability of Mixture Density Network to predict runtime of HPC applications in clouds. We started by explaining . We then designed two hybrid generative approaches to achieve more accurate prediction for HPC application runtimes in the cloud. In our hybrid approaches, automated model selection and model switching decreased the model bias and facilitated more accurate prediction of runtimes for the HPC application submitted to cloud.

We compared our prediction approaches with widely used approaches on available public HPC traces. Our MDN models improved over the accuracy of existing prediction approaches. Our MDN approaches can be used for more general problems of resource usage prediction and improve the performance and efficiency of distrbuted systems.

Chapter 7

Predicting CPU Usage with Deep Recurrent Neural Networks

Ensuring sustainability in the competitive cloud computing market requires cloud providers to incorporate more efficient resource management techniques to minimize their resource usage while providing competitive quality of service to customers. This level of efficiency is not possible without smart allocation of virtual machines on physical resources. Therefore, the future data centers should provide an unprecedented level of flexibility in resource management that requires intelligent decision making and scheduling of virtual machines. To that end, predictions of future resource consumption of virtual machines on individual virtual machine trace level is required. Accurate resource consumption forecasting has received attention in recent years, however, it has proven to be a difficult problem. We propose attention-based LSTM deep neural networks for long term prediction in individual CPU consumption of virtual machine traces. We investigate attention-based LSTM architectures and provide more accurate long term prediction for individual VM consumption. We train a deep LSTM network on CPU usage traces of 39000 virtual machines. We propose structural attention models appropriate for VM traces to improve LSTM prediction ability. To demonstrate the effectiveness of our approaches, we used our approaches to predict real-world workloads from Microsoft Azure Public Dataset. Our experiments demonstrate a considerable improvement in prediction error over currently used approaches for workload prediction.

7.1 Introduction

Data centers consumed about 1.3 percent of the worldwide electricity in 2012, and this fraction will grow to 8% by 2020 (54). There are increasing environmental concerns to reduce energy consumption and CO2 emissions in industry (85). Cloud providers such as Microsoft Azure, Amazon Web Services(AWS), and Google Cloud Platform (GCP) virtualize resources to allow consumers to access resources on a pay-as-you-go basis. Often, cloud providers offer a selection of virtual machines with a certain

amount of resources, including CPU, memory, etc. The customer purchases her choice of the virtual machine, and the cloud provider guarantees a level of quality of service (QoS) via service level agreement (SLA).

Cloud providers need to enable proactive resource provisioning to increase their revenues. One important observation is the fact that customers don't use all the resources they request. Trying to predict the actual resource consumption by each virtual machine and using it as a base for proactive elastic resource provision will improve efficiency. Cloud providers need to predict increases and decreases in resource consumption in individual VM level for efficient scheduling of workloads on resources in their clusters. Workload prediction plays a key role in proactive provisioning approaches and has been characterized as a hard problem (77).

The main objective of this paper is to study the effectiveness of sequence model deep learning approaches for long-term prediction of individual VM workload resource consumption in cloud platforms. In particular, we consider the problem of predicting future CPU consumption by virtual machines based on previous VM traces with attention-based Long Short Term Memory(LSTM) neural networks. One problem with encoder-decoder networks is their performance will deteriorate rapidly as the length of the input sequence increases. In time series analysis, this could be a concern since we usually expect to make predictions based on a relatively long segment of target series as well as driving series. The novelty in our work is that we train attention-based encoder-decoder networks to effectively predict future sequences of CPU usage for individual VM traces. We implement input-based attention to extract useful correlations with driving-traces. We also implement temporal attention mechanism to select relevant information stored in LSTM hidden units across all time steps.

In this work, we study the effectiveness of increasingly popular recurrent deep learning approaches for predicting virtual machine resource consumption (88). We will achieve this by training a *Long Short-Term Memory* (LSTM) recurrent neural networks on time series extracted from the log of previous traces. The LSTM has been found extremely successful in many applications with sequential data, such as unconstrained handwriting recognition (63), speech recognition (61) and machine translation (8). Recently, LSTM is applied for aggregate CPU usage on host servers (131), (42) and (65). In this chapter, we focus on using LSTM for predicting CPU usage in individual workload traces. We illustrate the feasibility of our approach by training a model on Microsoft Azure traces from $35,941$ deployments from 5,958 subscribers during a period of 30 days (azu).

In this paper, we predict the CPU usage by individual VM traces. The aims of this research are to :

- To investigate the accuracy of Recurrent Neural Networks for predicting future CPU usages of individual VM traces in comparison to traditional prediction methods.

- To study appropriate recurrent neural network structure and hyper-parameters

in order to achieve accurate prediction of CPU usage in individual VMs.

- To determine how far in the future the Recurrent Neural Network can accurately predict the CPU usage traces.

We use the sequence-to-sequence time series prediction to predict CPU consumptions across virtual machines. For this purpose, we train encoding decoding LSTM networks with multivariate inputs that are able to remember autocorrelation between sequence patterns as well as the correlation between input context (exogenous variables including additional features about time series) and sequence patterns.

This paper is organized as follows: We present the related work and the problem statement in Section 7.3, we explain our proposed approach in detail in Section 7.4. We then present our experimental setup and results in Section 7.5. Finally, we will conclude our work in Section 7.6.

7.2 Related Work

Different approaches are proposed to predict VM CPU usage. In (59), authors consider two different possible models to predict VM traces: signature-driven and state-driven models. They consider a cyclic pattern for a newly deployed VM and switch to state-driven model if they fail to find a signature after several resource consumption reports from the specific workload. In (105) each workload trace is decomposed into several wavelet signals and perform prediction for each signal separately. In (34), authors cluster VM trace patterns into several clusters called workload categories and use a stochastic model to predict CPU demand for each of the workload categories. A little different from most of the previous work, (80) identify groups of VMs that show recurring patterns. They train hidden Markov Model that utilizes temporal correlations in co-cluster patterns. Inspired by their work, we propose using attention-based recurrent neural networks to find co-clusters and extract the correlation between patterns automatically. In fact, recurrent neural networks have been proposed before for the different problem of aggregate workload CPU prediction before (65; 131; 42). In (42), authors proposed the usage of recurrent neural networks for prediction of host CPU utilization. In (65), the cloud overall CPU utilization is predicted with bidirectional multivariate input LSTM networks. The problem of predicting aggregated workload is an easier problem first because only a single aggregated trace is considered instead of various individual VM traces with various patterns. Second, the aggregate trace has fewer noise (80).

Recurrent neural networks are specifically efficient for sequential data and are appropriate for cloud virtual machine consumption trace time series. More specifically, we will study Long Short-term Memory deep learning networks and show how they outperform existing approaches for predicting aggregate CPU usage by virtual machines as well as resource usage for individual workloads. CPU consumption at the level of the host has been studied extensively in the past. For this purpose, the aggregated trace of CPU usages is studied as a single time series, and future values are

predicting based on the past and current values. In (59), authors use Markov Models to model resource usage on the host. In (105), authors use wavelet transformation of CPU usage trace as input to the Markov model to predict future values. (159) apply ARIMA to predict future values of for future values. (25) uses neural networks to predict future resource usage. We propose using attention-based LSTM recurrent neural networks as an end to end tool that extracts similarities between subsequences and finds correlation and auto-correlation among these subsequences by the help of LSTM gates parameters of which will be learned with back-propagation through time. We consider attention-based LSTM encoder decoders to make sure important information in hidden layers of LSTM network is not lost (8; 116). In addition to existing dual-level attention model for time series prediction, we also design a new attention model biases based on the structural consistency introduced by (80).

7.3 Background

7.3.1 Structure-based clustering and alignment

Finding repetitive patterns have been a common approach for workload characterization. Most of the existing work find a mapping between extracted features and a set of characteristics and use the mapping to predict the characteristics of target traces (7; 151). On a set of sample workloads (7) uses workload profiling to extract features corresponding events visible to virtual machine manager in each specific window of They use regression models to characterize the target workloads after training their regression models. However, these studies are concerned with statistically understanding and reproducing computing tasks (e.g., MapReduce tasks) scheduled on a cloud. Different from these works and inspired by some previous work on gene expression data (92), Khan et al. propose co-clustering VM traces and using Gaussian Hidden Markov Models (GHMM) (16; 113) to predict future patterns of VM consumption (80). Finding repetitive patterns and mining the correlation between the patterns have proved to be an effective approach in predicting individual VM CPU usage trace (26; 80). Finding both autocorrelations between different windows of a single trace as well as the correlation between windows in different traces help us to predict the values of CPU usage in future time steps. In (80) co-clustering of subsequences in traces to characterize the most repeating subsequences and then apply hidden Markov Models to characterize the temporal correlations in the discovered clusters and use these pieces of information to predict variations of VM workload patterns. Khan et al. define the consistency measure as:

$$const(trace) = \min_{j \in J} \frac{(\sum_{i \in I}(1 - |a_{ij} - a_j|/d))}{|I|}. \tag{7.1}$$

7.3.2 Recurrent Neural Network for load prediction

With the recent advancements in processors and the advent of deep neural networks, the application of deep learning approaches has increased substantially (107; 22; 88). The multi-layer neural network has been successfully used for predicting CPU loads on servers in distributed system (45). Among neural network architectures, the recurrent neural network is the most appropriate for predicting subsequences of traces.(42) use a recurrent neural network to predict one step ahead of host load in the cloud. As one problem with RNNs are the fact that information from multiple steps in the past will be lost because of multiple differentiation of the cost function in the optimization process, long short term memory (LSTM) have been proposed. In LSTMs self-loops conditioned on the context allows the network to remember information from the past selectively and avoids loss of useful information (57; 69). For these reasons (131) apply LSTM for multi-step prediction of hosts in the cloud. In (65), authors propose multi-variate bidirectional LSTMs for predicting aggregate utilization of workload in the cloud. They consider additional features including memory usage as input and incorporate more recent bidirectional LSTM (62).

7.3.3 Encoder-Decoder LSTM for sequence prediction

Recurrent neural networks are specially designed for sequential data (124; 96). In addition to connections between different layers of the neural network, RNNs have edges between values at different time steps. In training the RNN, the weights on temporal edges as well as layer edges are optimized. Back-propagation in feedforward networks moves backward from the final error through the outputs, weights, and inputs of each hidden layer, assigning those weights responsibility for a portion of the error by calculating their partial derivatives $-\partial E/\partial w$, or the relationship between their rates of change (68). Those derivatives are then used by the learning rule, gradient descent, to adjust the weights up or down, whichever direction decreases error.

In LSTM as a specific type of RNN, back-propagation goes through time steps as well as network layers. During training, LSTM learns what to remember and what to forget from the past time steps. As input size is large for LSTM, deep learning networks are usually trained in batches. Also, back-propagation is usually performed several times on the training set. Each complete run-through training set is called epochs. The choice of batch size, epochs, as well as training window, impacts the prediction power of the LSTM network. Usage of recurrent neural networks has been proved to be effective for server load prediction (42; 131).

Here, we present the mathematical formulation through the flow of information in Cell State in LSTM (70). For the simplicity of exposition, we refer to LSTMs as operating on a sequence that contains vectors $x(t)$ with the time step index t ranging from 1 to τ. Below is the formulas for forget gate, f_t. Matrices W_i, W_f, W_c, W_o represents the appropriate weight matrices. The vectors b_i, b_f, b_c , b_o denote the corresponding bias vectors. A sigmoid function is used to calculate the activation of

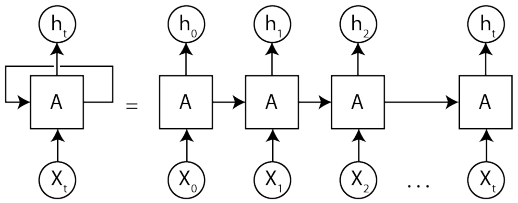

Figure 7.1: Each LSTM unit has self loop (left). The unrolled self loop is demonstrated on the right.

the forget gate f_t:

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f) \tag{7.2}$$

In the second step cell state determines what new information should be stored in the cell state. To start with, a sigmoid layer named the input gate layer decides which information should be updated. Then, a *tanh* layer creates a vector $\tilde{C}_t$ of new candidate values to be updated in the next state.

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_f) \tag{7.3}$$

$$\tilde{C}_t = \tanh(W_C.[h_{t-1}, x_t] + b_C) \tag{7.4}$$

Next, LSTM updates the old cell state $\tilde{C}_t$ into the new cell state C_t. C_{t-1} is multiplied by f_t to remove non-helpful information from old cell. $\hat{y}_M$ and f_M present the consumption component of previous time step and the features of this time step, respectively.Then it computes $i_t * \tilde{C}_t$. There are the new candidate values, scaled by how much information should be updated in each state value.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{7.5}$$

Lastly, LSTM needs to decide the output. This has two parts:A sigmoid activation function layer as output gate is used to filter the cell state firstly. Then, the cell state is put through tangent hyperbolic ($tanh()$) and is multiplied by the output o_t to calculate the desired information.

$$O_t = \sigma(W_0.[h_{t-1}, x_t] + b_0) \tag{7.6}$$

$$h_t = O_t * tanh(C_t) \tag{7.7}$$

The value of weight matrices, W_i, W_f, W_c, as well as bias vectors, b_i, b_f, b_c , b_o will be determined through an optimization procedure called back-propagation through time in model training stage.

For predicting long term dependencies in time series data, sequence to sequence LSTM networks are proved to be effective (57). As shown in Figure 7.3, in these networks, the encoder calculates an encoding of the input and passes it to the decoder. The encoding is, in fact, the hidden state of the last time step in encoder LSTM. The

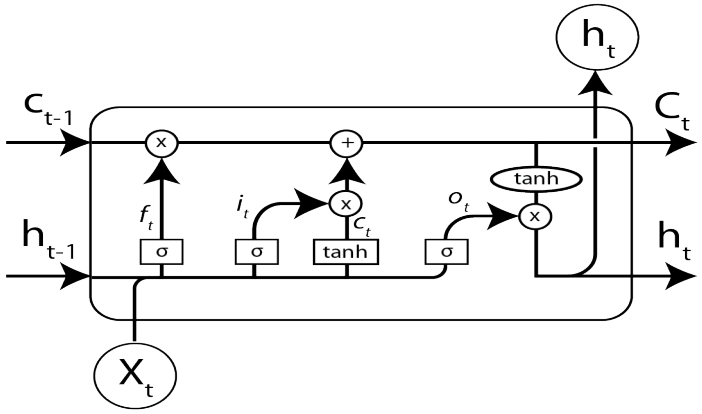

Figure 7.2: Anatomy of LSTM network.

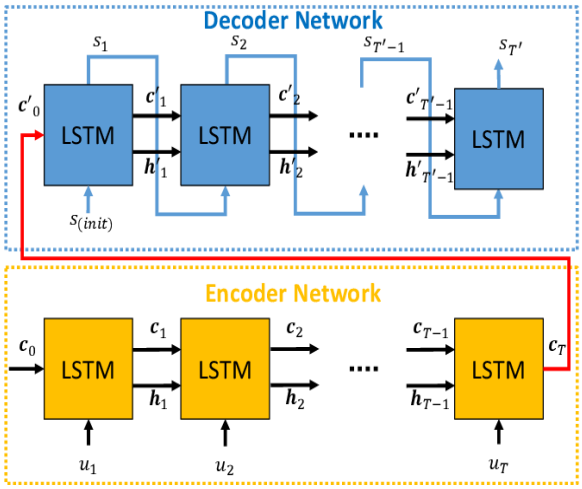

Figure 7.3: Sequence prediction with encoder-decoder LSTM

decoder generates the prediction of the target sequence using the encoding and target sequence from the last window of time steps.

7.4 Our Proposed LSTM model for predicting individual VM patterns

In this section, we describe our proposed prediction methodology based on the fundamentals of LSTM recurrent neural networks presented in the previous section. We propose to tailor existing attention-based LSTM approaches for sequence-to-sequence prediction to build a predictive model. The predictive model is trained using available VM consumption traces and predicts future consumption patterns for each given individual VM trace. We train a model with CPU Consumption traces of 80,000 individual VMs. As it is observable in available public VM traces and also pointed out in (106), CPU usage for each virtual machine is not constant and changes over VM lifetime. The objective is to accurately predict multiple time steps in the future for an individual virtual machine, given historical CPU usage for that specific virtual machine. The CPU usage for the previous M time-steps as well as other eight features listed in Table 7.2 are given as input to the model. The proposed LSTM model predicts the CPU usage in each of the next M time steps.

Based on the works of (80), we extend the existing attention-based encoder-decoder LSTM models on making them more effective for VM trace data. We design encoder-decoder LSTM networks that take advantage of VM trace pattern matching techniques. For this purpose, we improve existing encoder-decoder LSTM models for trace prediction in two ways: we expand the LSTM input with additional pattern matching based features as well as frequency features. Also, we design specialized attention-based LSTM models that determine the attention to the specific hidden state of a decoder based on the pattern matching structure of input sequence and predicted subsequence of the output sequence. We propose attention-based recurrent neural network models to predict individual VM traces. Different attention mechanisms have been introduced in the fields of computer vision (127), neural machine translation (97), image captioning (155; 153), and speech recognition (31). We picked the most useful attention mechanisms for our multivariate trace prediction problem and improved them based on the trace prediction literature to achieve more accurate predictions. We considered applying attention model both for different dimensions of input features as well as temporal attention. Attention is a mechanism in recurrent neural networks that weights some inputs or hidden states at specific time lags in encoder more than others to achieve better prediction (74).

7.4.1 The attention mechanism and structural bias

Attention models in recurrent neural networks propose to adaptively focus on a specific part of the inputs of the recurrent neural network to improve the quality of

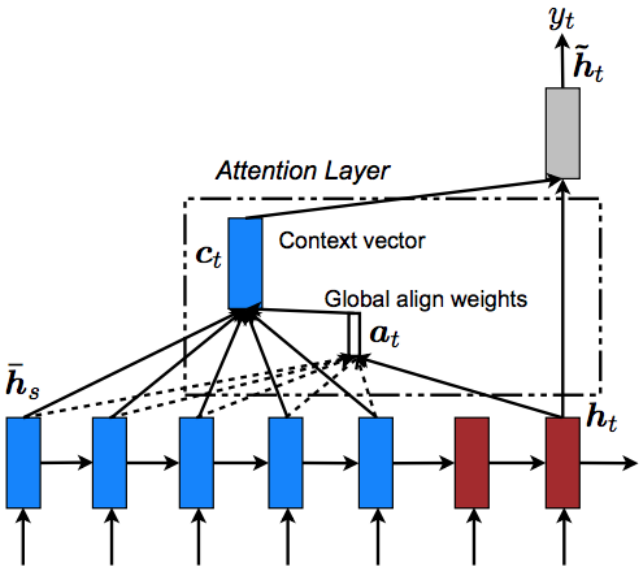

Figure 7.4: The mechanism of attention model (97).

learning. As reviewed in Section 7.3 and shown in Figure 7.2, LSTM units are functions of three sets of inputs: input time series (X_t), hidden states from the previous time step (h_{t-1}) and cell state from previous time step CS_{t-1}. The earlier attention models focused on temporal attention (8). Temporal attention model considers all hidden states of encoder instead of the last hidden state and creates a context vector by creating a weighted sum of hidden states of the encoder (8). Temporal attention enables search over different alignments of source and target sequences to achieve more accurate prediction (146). The temporal attention models consider all hidden states of the encoder instead of considering only the last hidden state of the encoder to create the context vector. Later attention models also considered different weighting dimensions of the multi-dimensional input time series. As multi-variate inputs are often used for time series prediction, authors in (116) have proposed an additional input attention model that weights the different dimensions of the input time series based on their effectiveness in minimizing the cost function. The input attention model considers the weighted sum of input features and allows the model to use available input features more selectively based on the context.

The general framework for building an attention model is to design a scoring function that measures the relevance of each specific part of the input and the current hidden state of the encoder. Having a scoring function, the weight for each hidden layer of the encoder is determined and used for calculation of the context vector. The temporal attention mechanism is shown in Figure 7.4. We review the general framework of attention to provide intuition and background theory for the specific attention mechanisms presented following the notations in (97). In sequence-to-sequence prediction with a recurrent neural network, the context vector transfers useful in-

Table 7.1: Different scoring functions considered for attention model.

scoring function name	equation
inner product	$h_t^T \bar{h}_s$
general	${h_t}^T W_a \bar{h}_s$
concatenation	$v_a^T tanh(W_a[h_t; \bar{h}_s])$

formation from encoders to be used in encoders. Attention mechanism modifies the construction of context vector to improve the encoding-decoding sequence prediction power. The context vector for temporal attention model denoted as c_t, allows the decoder to capture global information rather than solely infer based on one hidden state. To build context vector c_t, for a single CPU usage trace prediction at time step s, y_s, the model loops over all encoders' states, $\bar{h}_s$, to compare the current target hidden state (decoder's hidden state) and each of the source hidden states (different encoders' states from 1 to S) as in Equation 7.9. The comparison calculates a similarity score that we denote as $score(h_t, \bar{h_s})$. To calculate the probability distribution of y_t as in Equation 7.11 conditioned on target states, the scores are normalized to sum to one over all S values of encoder hidden states as shown in Equation 7.8. To make the attention mechanism trainable, weights are used as parameters to enable learning appropriate context vectors. The calculation of attention is shown in Equation 7.10. The attention vector for decoder

$$\alpha_{ts} = \frac{exp(score(h_t, \bar{h_s})}{\sum_{s'=1}^{s} \exp score(h_t, \bar{h}_{s'})} \tag{7.8}$$

$$c_t = \sum_S \alpha_{ts} \bar{h}_s \tag{7.9}$$

$$\tilde{h}_t = f(c_t, h_t) = tanh(W_C[c_t; h_t]) \tag{7.10}$$

$$p(y_t | y_{<t}, x) = softmax(W_s \tilde{h}_t) \tag{7.11}$$

As the goal for the model is to focus on more relevant information, a scoring function is employed to formulate the similarity of sth hidden state in encoder and the current encoder. The differences in attention models are mainly in the way this similarity is formulated.

The different scoring functions are listed in Table 7.1.

In this work we adopt the dual attention model from (116) and add structural VM trace pattern bias to make it more effective for VM trace prediction. In the first stage of the dual attention model, the scores for different dimension of the input vector is calculated to determine the appropriate weighting of different dimensions. Given the k-input series $x^k = (x_1^k, ..., x_T^k)^T \in R^T$, an input attention mechanism is constructed via a multilayer perceptron, by using h_{t-1} and the cell state s_{t-1} in the encoder LSTM unit with:

$$e_t^k = v_e^T tanh(W_e[h_{t-1}; s_{t-1}] + U_e x^k) \tag{7.12}$$

and

$$\alpha_t^k = \frac{\exp e_t^k}{\sum_{i=1}^{n} \exp e_t^i} \tag{7.13}$$

where $v_e \in R^T$, $W_e \in R^{T \times 2m}$ and $U_e \in R^{T \times T}$ are parameters to learn. A softmax function is applied to e_t^k to ensure all attention weights sum to one. The parameters of the attention model can be learned with other components of our LSTM model. With these attention weights effective covariate series are extracted adaptively as follows.

$$\tilde{x}_t = (\alpha_t^1 x_t^1, \alpha_t^2 x_t^2, ..., \alpha_t^n x_t^n)^T. \tag{7.14}$$

Then the hidden state at time t can be updated as

$$h_t = f_1(h_{t-1}, \tilde{x}_t). \tag{7.15}$$

Where f_1 is an LSTM unit that can be computed according to Equation 7.3 with x_t replaced by the newly computed $\tilde{x}_t$.

Another issue is that for time series prediction, we want the model to consider autocorrelations and correlations from far in the past, and we need the model to choose previous lags selectively. Temporal attention can bring useful information from a distant past to the current RNN cell. Attention models select the most pertinent piece of information, rather than using all the available information large part of which is irrelevant to compute the neural response. To predict the output $\hat{y}_T$, we use a decoder to decode the encoded input information. However to avoid degradation of encoder-decoder network with the increase in length of the sequences, a temporal attention mechanism is used in the decoder to adaptively select relevant encoder hidden states across all time steps. The attention weight of each encoder hidden state at time t is calculated based upon the previous decoder hidden state $d_{t-1} \in R^p$ and the cell state of the LSTM unit $s'_{t-1} \in R^p$ with

$$l_t^i = v_d^T tanh W_d[d_{t-1}; s'_{t-1}] + U_d h_i \quad q \leq i \leq T \tag{7.16}$$

and

$$\beta_t^i = \frac{\exp l_t^i}{\sum_{j=1}^{T} \exp l_t^j} \tag{7.17}$$

Where $[d_{t-1}; s'_{t-1}] \in R^{2p}$ is a concatenation of the previous hidden state and cell state of the LSTM unit. Where $v_d \in R^m$ and $W_d \in R^{m \times 2p}$ and $U_d \in R^{m \times m}$ are parameters to be learned. The attention weight β_t^i represents the importance of the i encoder hidden state for prediction. Since each encoder hidden state h_i is mapped to a temporal component of the input, the attention mechanism computes the context vector c_t as a weighted sum of all the encoder hidden states $\{h_1, h_2, ..., h_T\}$,

$$\sum_{i=1}^{T} \beta_t^i h_i. \tag{7.18}$$

After calculating the weighted summed context vectors, context vectors are combined with the given target series $(y_1, y_2, ..., y_{T-1})$:

$$\tilde{y}_{t-1} = \tilde{w}^T [y_{t-1}; c_{t-1}] + \tilde{b}, \tag{7.19}$$

where $[y_{t-1}; c_{t-1} \in R^{m+1}$ is a concatenation of decoder input y_{t-1} and c_{t-1}. Parameter $\tilde{w} \in R^{m+1}$and $\tilde{b} \in R$ map the concatenation to the size the decoder input. The newly computed $\tilde{y}_{t-1}$ are used to update of the decoder hidden state at time t:

$$d_t = f_2(d_{t-1}, \tilde{y}_{t-1}) \tag{7.20}$$

We choose the nonlinear function f_2 as an LSTM unit. Then d_t can be updated as:

$$f'_t = \sigma(W'_f[d_{t-1}; \tilde{y}_{t-1}] + b'_f) \tag{7.21}$$

7.4.2 Specialized structural bias attention mechanisms

Similar to the work in (32) where authors design specialized bias models for neural machine translation based on literature in statistical machine translation, we design structural bias for trace prediction based on literature in statistical trace characterization (80).

Temporal bias We include a temporal bias through redefining the pre-normalised attention scalars in Equation 7.10 as follows:

$$\alpha_t = f(c_t, h_t) = tanh(W_C[c_t; h_t]) + W_1\psi(j, i) \tag{7.22}$$

Consistency pattern bias To improve the effectiveness of attention model for predicting workload patterns, we add a measure of consistancy on the literature on co-clustering approaches for workload prediction. This bias is based on the idea that one characterization of CPU traces is their consistency measure (80). We define the consistency bias attention between ith element of the source subsequence and $(j-1)$th element of the target subsequence as how close each of the elements in their corresponding subsequences are to these values. We also add the value of $j-1$ as the number of elements from the target subsequence considered for the calculation of the consistency:

$$\phi(j, i) = [\frac{(\sum_{0 \leq k < j-1} |a_k - a_{j-1}|)/d}{|I|}, \frac{(\sum_{l \in J} |a_l - a_j|)/d}{|J|}, |J|]^T. \tag{7.23}$$

$$\alpha_t = f(c_t, h_t) = tanh(W_C[c_t; h_t]) + W_1\psi(j, i) + W_2\phi(j, i) \tag{7.24}$$

7.4.3 Additional input features to improve prediction accuracy

As discussed earlier in this section, we add multiple features to individual CPU time-series to capture characteristics of the CPU usage for individual VMs. Using LSTM with multidimensional input features gives our model the power to predict a wide variety of virtual machines without using additional clustering and similarity calculation. The feature series considered are listed in Table 7.2. A set of effective positional features are CPU reading values from time lags with specific depth in the past. There are three most important points in the past that we use as a fixed weight attention feature (taking into account long-term seasonality): 1) 1 hour ago, 24 hours ago,

Table 7.2: Features considered for our prediction model.

feature group	feature names	number of features
original CPU trace	y_t	1
statistics of past data e	mean, std	1
seasonality features	hod, dow	2
autocorrelation	lag1, lag2, lag12	3
lagged trace values	ld, lw	2

2) One week ago. Recurrent neural network prediction with multi-dimensional input time series is not accurate and the model cannot choose which of the covariates to focus for making the prediction (116). To avoid the curse of dimensionality and improve the performance of multi-variate prediction model, we use input attention model proposed in (116) to pick the most relevant features for prediction of each sequence. This is done as the first stage of the dual-stage attention model that we adopt from (116). At encoding step, we incorporate input attention model to choose the most relevant feature time series for each sequence.

7.5 Experimental Results and Discussion

In this work, we focus on predicting resource consumption by workloads in production clusters. The production workloads include internal vendor VMs comprising of research and development and infrastructure management workloads as well as services provided to the third party customers, including communication, gaming, data management (33). The original trace used in (33) contains information about every VM running on Microsoft Azure for three month period from November 16, 2016 to Feb 16, 2017. The dataset also contains information on resource requests submitted by each subscriber in each deployment. The VM resource usage measurements are reported with five-minutes (300 seconds) resolution. The data used to train the encoder-decoder LSTM networks in our research are extracted from Microsoft Azure public workload dataset. The Microsoft Azure public dataset contains a representative subset of 30 consecutive days of traces of the first-party virtual machine workload (VM) of Microsoft Azure in one of its geographical regions. The trace is a sanitized subset of the Azure VM workload described in (33). We extracted time series for maximum and average CPU consumption for each of the 2,013,767 individual virtual machines. Fig 7.5 shows virtual machine traces for three VMs during the first week of this period. While temporal patterns are observable in time series of different VMs, we can see that each VM has a specific pattern and a single simple model cannot predict future values of individual VMs accurately. There are two main information sources for prediction:

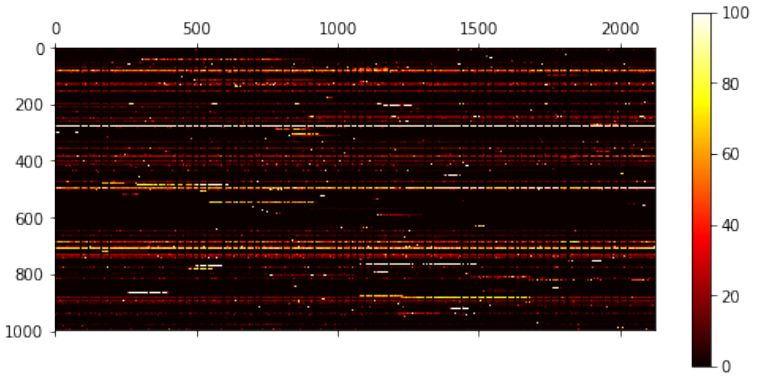

Figure 7.5: Heatmap of max CPU usage for 3000 VMs from Microsoft Azure public dataset.

1. Local features: If we see a trend, we expect that it will continue (AutoRegressive model), if we see a consumption spike, it will gradually decay (Moving Average model), if we see more CPU consumption on holidays, we expect to have more CPU consumption on holidays in the future (seasonal model).

2. Global features: If we look to autocorrelation plot, we'll notice strong week-to-week and day-to-day autocorrelation.

We illustrate heatmap of 3000 virtual machines in Microsoft Azure in Fig 7.5. Each row on the heatmap represents a single virtual machine. Darker color represents lower CPU consumption and brighter colors show higher CPU consumption by each virtual machine.

7.5.1 Data Preparation and feature extraction

The original data in public Azure dataset is available in VM CPU readings in comma-separated vector (CSV) format. The tables are indexed by virtual machine anonymized IDs. Each row of the table contains information on resource usage for specific VM ID at a specific timestamp. The original CPU readings are available publicly in 125 tables of $1GB$ each. We applied database table manipulation techniques using Python Pandas package to extract individual time series for each virtual machine. From the CPU reading tables, we built VM consumption matrices containing individual VM consumption traces using data management techniques. For each VM ID, we extracted the time series of maximum CPU consumption and stacked the time series to form a matrix of $2,013,767$ rows. Before using our data for building the prediction models we did a couple of common data preprocessing to achieve better prediction models. We removed all zero columns - time-steps that all CPU readings for all VMs are zero. We also applied local smoothing to alleviate spikes in the data. For predicting overall maximum CPU consumption, we created a time series composed of the sum over columns of the VM consumption matrix. For the remainder of the

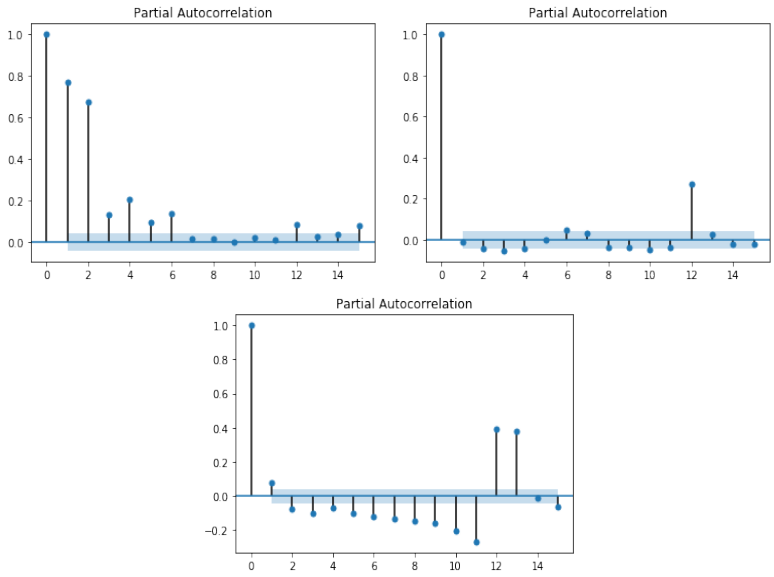

Figure 7.6: Partial auto-correlation of three virtual machines.

paper, wherever we talk about VM consumption, maximum consumption is intended except otherwise is specified. We hand-picked a few temporal and seasonal features to consider as extra inputs in addition to the original max CPU time series values that improved our model prediction accuracy. As LSTM does a good job to discover and learn features from the lagged CPU usage traces on its own, feature engineering is not necessary. However, as individual VM prediction is a hard problem, we included 8 additional features to improve the prediction accuracy of the model. Our experiments approve the effectiveness of additional features. We considered the following eight features in addition to the original time series improve our prediction as listed in Table 7.2: We extracted Mean and standard deviation for CPU usage in the past as we saw high load and low load time series have different patterns. We also added two seasonality features, an hour of the day and the day of the week to capture daily and weekly seasonality. Through analysis of autocorrelation plots of different virtual machines, we realized the VMs have different patterns of autocorrelations. The partial autocorrelation plots are shown in Fig 7.6. As first, second and 12th autocorrelations were the most distinctive values we considered these values as additional features. These three features capture consecutive autocorrelation strength as well as the strength of correlation with 10 minutes ago CPU utilization reading as well as that of last hour. We also realized, adding the value of max CPU load for the exact time in the previous day improved the prediction accuracy.

Our input data is a nine-dimensional time series consisting of original max CPU values as well as eight other features listed in Table 7.2. The length of each of the time series is 8640 timestamps. Our dataset differs from an ordinary machine learning dataset as the data is collected over time. Generally, machine learning datasets are collections of observations with no consideration of time. In those problems,

predictions are made for new data when the actual outcome may not be known until some future date. The future is being predicted, but all prior observations are almost always treated equally. Perhaps with some very minor temporal dynamics to overcome the idea of *concept drift* such as only using the last year's observations rather than all data available. A time-series dataset is different. Time series adds an explicit order dependence between observations: a time dimension. This additional dimension is both a constraint and a structure that provides a source of additional information. Time series prediction involves taking models to fit on past data and using them to predict future observations. Time series based prediction models often learn the degree of autoregression and seasonality from the past and use the learned model for future data. We implemented the presented prediction approaches on Azure Public Data Set introduced in (33) to show the effectiveness of our proposed solutions. The trace contains a representative subset of the first-party Azure VM workload in one geographical region. The dataset contains information on CPU usage for $2,013,767$ VMs during 30 days. The VM resource usage measurements are reported with five-minutes (300 seconds) resolution.

7.5.2 Using LSTM to predict CPU consumption

The objective is to predict individual VM CPU consumption multiple time steps in the future, given historical aggregate CPU usage. The input to the model can be expressed as

$$y = \{y_0, y_1, ..., y_{M-1}\} \tag{7.25}$$

where y_t is the aggregate CPU measurement for time step t. The predicted CPU consumptions can be expressed as:

$$\hat{y} = \{\hat{y}_M, \hat{y}_{M+1}, ..., \hat{y}_T\} \tag{7.26}$$

The input dataset was the aggregate CPU usage by all virtual machines in 30 days. We chose a deep LSTM structure through experimentation with the training set and validation set data. The architecture is shown in Fig 7.3. The input to the network are subsequences of M aggregate CPU readings. To train the model, back-propagation through time is used. We chose Mean Squared Error as objective function, as shown in Equation 7.27.

$$J_\theta = \sum_{M}^{T} (y_t - \hat{y}_t)^2 \tag{7.27}$$

As we observed in Microsoft Azure dataset and also pointed out in previous literature, CPU usage for each virtual machine is not constant and changes over VM lifetime (106). The objective is to accurately predict multiple time steps in the future for an individual virtual machine, given past CPU usage for that specific virtual machine. The maximum CPU usage for the previous M time-steps as well as other eight features listed in Table 7.2. LSTM model predicts the maximum CPU usage in the next M steps. Training a model that is able to predict individual VM CPU consumption traces is a hard problem if we take into account variance in different virtual machine

consumption patterns. Some previous works, consider multiple training models and assign the most appropriate model by looking into the input feature vector and finding the cluster it is most similar to. Instead of clustering and categorizing virtual machines, we calculated several additional features for each VM to capture the characteristics of that virtual machine including autocorrelation pattern, the maximum value and standard deviation in available past time steps. In fact, using LSTM with multi-dimensional input features gives our model the power to predict a wide variety of virtual machines without using additional clustering and similarity calculation. For further improvement of our individual VM prediction LSTM model, we used attention mechanism. There are three most important points in the past that we use as a fixed weight attention feature (taking into account long-term seasonality): 1) One hour ago, 24 hours ago, 2) One week ago. To perform training, we considered 24 hours windows for each time series and applied zero paddings for shorter virtual machines and random cutting for longer ones. As we are training our model with time series related to different VMs, there is a high probability of abrupt changes in loss function when performing SGD. We apply gradient clipping for better convergence of loss function (114).

In the training stage, the weight values described in the previous section are determined using back-propagation through time (BPTT) with regard to J_θ. The gradient-based optimization approach that worked well for this specific problem was ADAM. ADAM is known for being efficient and appropriate for large problems, and it usually requires minimal tuning (81). For the loss function, we used the Mean Squared Error (MSE). To perform training, we considered 24 hours windows for each time series and applied zero padding for shorter virtual machines and random cutting for longer ones. As we are training our model with time series related to different VMs, there is a high probability of abrupt changes in loss function when performing SGD. We apply gradient clipping for better convergence of loss function (114).

We implemented encoder-decoder LSTM with attention model using Tensorflow 1.13 (5). We realized that the batch size of 288 (which is in fact 24 hours of every minutes CPU readings) and a window size of 12 works best. To study the effectiveness of attention models and structural bias, we repeated the experiments with ANN, encoder-decoder LSTM without attention mechanism and LSTM encoder-decoder model with one of the two attention models. We also compared our model with encoder-decoder LSTM with dual attention model without structural bias. In each of these configurations, 4 layer LSTM is trained on data for 24 hours of max CPU reading. The trained LSTM is used to predict the maximum CPU usage for the next 24 hours. We evaluate our prediction model by calculating and comparing the mean absolute percentage error (MAPE), root means squared error (RMSE), and mean absolute error (MAE) with existing methods. The mean absolute percentage error (MAPE), also known as mean absolute percentage deviation (MAPD), is a measure of predicting the accuracy of a forecasting method in statistics, for example, in trend estimation. It usually expresses accuracy as a percentage, and is defined by the

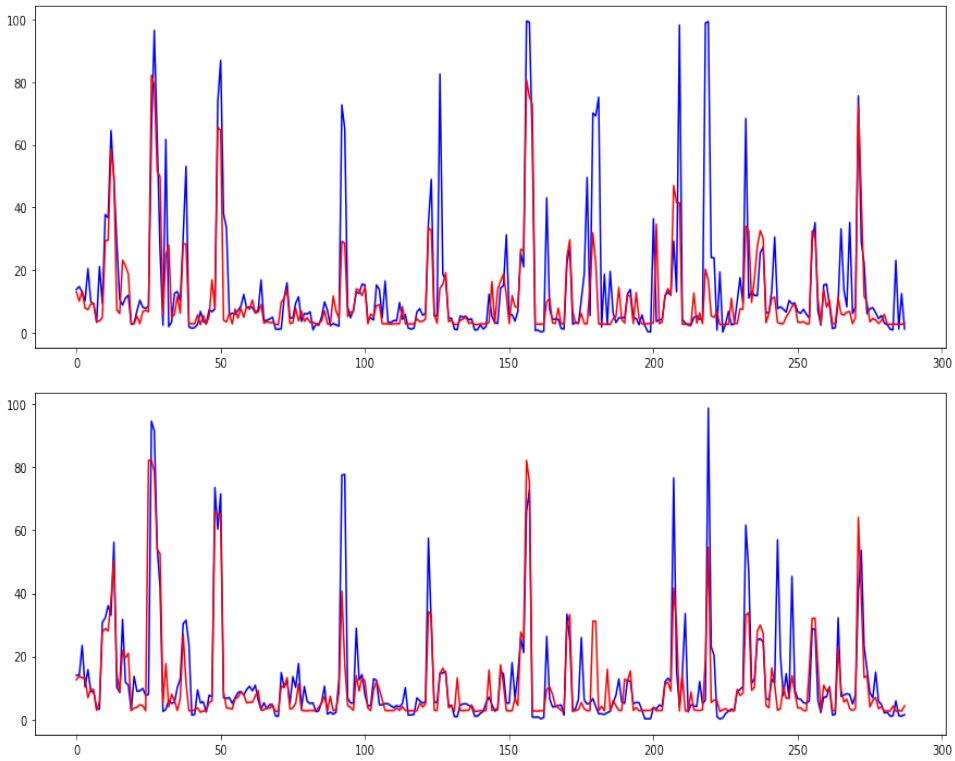

Figure 7.7: VM cpu utilization prediction for to VMs.

Table 7.3: Specifications of LSTM Network for Individual VMs CPU Consumption.

Training Fuction	Back-Propagation
Number of Hidden Layers	4
Nodes in Each Layer	12,10,10, 1
Batch Size	288
Sequence Length	12
Loss Function	MSE
Optimization Method	SGD with Gradient Clipping
Learning Rate Schedule	0.1, 0.01, 0.001
Gradient Clipping Value	5
Performance Goal	0.001

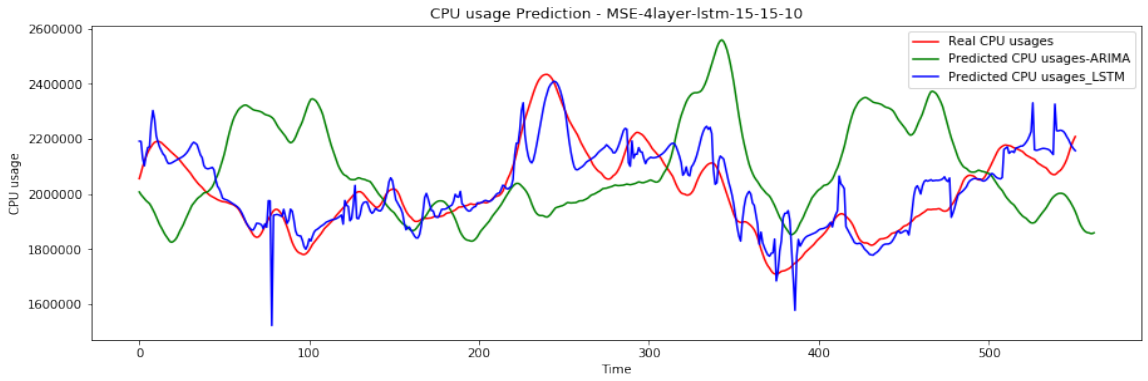

Figure 7.8: aggregated CPU usage prediction. LSTM is compared with ARIMA and ANN.

Table 7.4: Mean absolute percentage error for individual VM CPU consumption.

Approach	ANN	LSTM	LSTM+INP	LSTM+INP+TP	LSTM+INP+TP+bias
MAPE	29%	17%	16%	14%	11.5%

formula:

$$M = \frac{100\%}{n} \sum_{t=1}^{n} \left| \frac{y_t - \hat{y}_t}{y_t} \right|$$

where y_t is the actual value and $\hat{y}_t$ is the forecast value.

Compared methods: ANN (Artificial Neural Network): We considered 12 recent values of each of nine features. So, our the number of input features -or dimension of the input- for the ANN is 108. Our neural network architecture is similar to the network used in (73).

LSTM: We implemented encoder-decoder LSTM as described in Section 7.4. The window size is 12 time-steps, and the batch size is 288.

LSTM with input attention (LSTM+INP): In this configuration we added input attention model as described in Section 7.4 to the encoder-decoder model.

LSTM with temporal attention (LSTM+TP): In this configuration we added temporal attention as described in Section 7.4 to the LSTM model.

LSTM with dual attention (LSTM+INP+TP): In this configuration we added dual attention (temporal and input) as described in Section 7.4 to the LSTM model.

LSTM with dual attention and structural bias (LSTM+TP+INP+bias): In this configuration we added structural bias as described in Section 7.4 to the LSTM model with dual attention.

7.5.3 Discussion: Comparison with Existing Approaches for Workload Prediction

The disadvantage of traditional neural networks like all other non-sequential machine learning approaches is that those models are valid for a certain period only and temporal dependence of variables across different timestamps are not taken into account when building a predictive model. Traditional neural network approaches do not consider the temporal characteristics of input features in building the predictive model.

Recurrent Neural Networks have been proposed as neural networks with time series as input. So, in addition to learning a mapping between feature values and target values, they also learn the mapping between the previous values of the target value and its future values. These approaches are shown to be considerably more effective than other types of neural networks for time series data (96), (124). Recurrent neural networks architecture is composed of multiple states similar to Markov chains. In the process of training RNNs, back-propagation through time is applied to determine model weights. As one can imagine, long back-propagation can lead to exploding or vanishing gradients as a result of many extremely small or extremely large values multiplied together (69). To overcome this, a special group of RNNS called LSTM have been proposed (70). LSTMs are able to store information for a long period of time and are appropriate for our application.

LSTM can be thought of as a natural extension of well-studied ARIMA models, but much more flexible and expressive. LSTM is non-parametric, that greatly simplifies learning. Once we train our LSTM model, we are able to use the same model for the prediction of different test subsequences. Imagine working with different ARIMA parameters for more than 2 millions time-series for individual virtual machine CPU usages. Any exogenous feature (numerical or categorical, time-dependent or series-dependent) can be easily injected into the model. LSTM Sequence to sequence prediction predicts next values, conditioning on joint probability of previous values, including our past predictions. Use of past predictions stabilizes the model. It learns to be conservative, because error accumulates at each step, and an extreme prediction at one step can ruin prediction quality for all subsequent steps. For a complete review of LSTMs, we refer the readers to chapter 10 of (60). Deep Learning provides a more accurate prediction with the fine granularity of time-steps for mid-term as well as long-term future. There is no need for feature engineering parameter tuning for each individual time series.

Two different categories of approaches are used to predict time series:

1. Considering previous values as features and building input vectors from multiple recent values and applying common machine learning approaches to predict future values based on input features.

2. Applying time-series specific approaches such as ARIMA that automatically consider temporal autocorrelation with previous values of the time series.

The problem with the first category of approaches is the fact they treat each of the previous values equally. For instance, if we input three recent values as input to an ANN, the model treats the immediate past value similar to the value with lag 2. This leads to loss of valuable information about correlations consecutive lags. Recurrent neural networks purposefully designed to learn all these patterns from sequences.

ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration.

In predictions with ARIMA, a linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model.

One limitation of ARIMA approaches is the requirement of the parameter setting. We want to apply our prediction model to predict individual virtual machine traces. Using ARIMA, we need to figure out the value of p, d, and q for each VM. Additionally, As our CPU utilization trace may not be consistent throughout VM lifetime, we need to update the parameter p, d, and q for each prediction. Another issue is the fact that ARIMA builds a linear regression model on the processed lags and is not able to learn nonlinear patterns. Consequently, although ARIMA may give good predictions for the next couple of time series value, it is not able to give an accurate long-term prediction.

However, an LSTM recurrent neural network is able to build an end to end model with high generalization ability. With the help of gated nodes and the layered structure can predict future CPU utilization values for utilization traces of individual VMs without requiring manual parameter setting and parameter updating.

7.5.4 Feasibility of Our Prediction Model for Resource Management Systems

After training the model on training data, the trained model is deployable to predict next hour worth of max CPU consumption for 12 lags in next hour, given CPU usage data for 12 lags in the current hour. Applying the model to predict is, in fact, multiple matrix multiplications and takes less than a minute to calculate in an ordinary dual-core system. Our experiments show that the LSTM network is powerful enough to predict new sequences. In deployment, it is possible to monitor the performance of the prediction and retrain the model with newer data as soon as drift is detected.

7.6 Summary

In this chapter, we proposed LSTM approaches to predict maximum CPU consumption both for individual virtual machines as well as overall CPU consumption for large cloud platform data centers. For both problems, LSTM based architectures were trained and tested on real-world production trace data. For training LSTM

for individual VM CPU consumption, we used the dual-stage attention model and added CPU level consistency bias to achieve a generalized model with high accuracy prediction of individual VMs. For the optimization, we performed gradient clipping to avoid gradient explosion in back-propagation. We also included autocorrelation and frequency features to improve LSTM prediction ability with leveraging selective temporal dependencies. Our experimental evaluations showed our designed attention-based LSTM outperforms commonly used approaches in prediction accuracy. Our attention-based LSTM provides an accurate long-term and mid-term prediction for resource usages. We discussed why LSTM is more appropriate than other nonrecurrent deep learning approaches. We also discussed how LSTM is more accurate than ARIMA for midterm and long-term prediction of aggregate CPU consumption traces. Additionally, we studied the effect of attention mechanism to improve the encoder-decoder LSTM model for prediction individual VM CPU usage traces.

Chapter 8

Concluding Remarks

High Performance Computing is behind most of the recent advances in science and engineering. Researches performed in national labs are not possible without the enormous parallel interconnected supercomputers known as high performance systems. As discussed in Chapter 1, more advanced resource management, and application schedulers are required to provide the desired performance demands of customers while keeping the expenses of running HPC systems reasonable. In conclusion, we briefly highlight the major contributions and future research directions that can be built upon the outcomes of this research under three main categories including scheduling algorithm design, application runtime prediction, and applications resource usage prediction.

8.1 Scheduling Nonpreemptive Applications in Distributed Systems

Our first goal of this Ph.D. thesis was to study the theoretical problem of scheduling non-preemptive applications and propose effective and efficient scheduling algorithms. In order to achieve this goal, we have proposed SVF (Smallest Volume First) algorithm in Chapter 3.5. We proved that SVF's average completion time is a constant multiple of the optimal scheduler in the worst-case scenario. We also performed trace-based simulations to compare the performance (average completion time) and efficiency (CPU utilization) of SVF with existing scheduling approaches.

8.2 Runtime Prediction for HPC Workload

Our second goal of this Ph.D. thesis was to study the inaccuracy of runtime prediction for HPC applications, its impact on scheduling performance and propose solutions to improve the scheduling performance. For this purpose, we propose to solution for improving scheduling performance deterioration caused by inaccurate runtime.

Our first solution is a hybrid scheduling platform that arbitrates between two different scheduling strategies based on the predictability of application runtime. This solution is presented in Chapter 4. The choice of scheduling strategies is based on their sensitivity to runtime accuracy. Our extensive simulation experiments showed that plan-based scheduling strategies have high performance when accurate runtimes are available, but their performance highly degrades when job runtimes are inaccurate. While plan-based scheduling strategies are highly sensitive to runtime accuracy, backfilling strategies have lower sensitivity to runtime accuracy. Backfilling strategies mainly perform FCFS and then greedily pack unassigned jobs on remaining available resources based on the available estimation of runtime. As the initial FCFS stage of backfilling does not use the estimation of job runtime, these strategies are less sensitive to the job runtime accuracy. Our hybrid scheduling platform partitions waiting jobs into two groups of predictable and unpredictable and applied the appropriate strategy for jobs in each partition. To avoid resource fragmentation, we use resource sharing in our hybrid scheduling platform. Our experiments show that the Hybrid scheduling platform improves scheduling performance by 20% in the real-world scenario where average runtime accuracy is about 70%.

Our second solution for improving performance degradation because of inaccurate job runtimes is proposing more accurate runtime prediction approaches. For this purpose, we propose two new prediction approaches: one approach for predicting HPC runtimes in the Cloud is discussed in Chapter 5 and one for predicting runtimes for HPC workload on HPC clusters, discussed in Chapter 6. The difference between cluster and Cloud is that in the Cloud, we have more variation in workload features. For this purpose, we propose an adaptive online prediction approach based on linear state-space models (Kalman Filters).

For job runtime prediction in HPC clusters, we look into the distribution of workload data. As the workload is multi-modal, we propose Mixture Density network for HPC clusters.

8.3 Predicting CPU Consumption Patterns in Distributed Systems

To achieve the third goal of this Ph.D. thesis, we proposed LSTM approaches to predict maximum CPU consumption both for individual virtual machines as well as overall CPU consumption for large cloud platform data centers. For both problems, LSTM based architectures were trained and tested on real-world production trace data. For training LSTM for individual VM CPU consumption, we used the dual-stage attention model and added CPU level consistency bias to achieve a generalized model with high accuracy prediction of individual VMs. For the optimization, we performed gradient clipping to avoid gradient explosion in back-propagation. Moreover, we included autocorrelation and frequency features to improve LSTM prediction ability with leveraging selective temporal dependencies.

Our experimental evaluations showed our designed attention-based LSTM outperforms commonly used approaches in prediction accuracy. Our attention-based LSTM provides an accurate long-term and mid-term prediction for resource usages. We discussed why LSTM is more appropriate than other nonrecurrent deep learning approaches. Additionally, discussed how LSTM is more accurate than ARIMA for midterm and long-term prediction of aggregate CPU consumption traces. Additionally, we studied the effect of attention mechanism to improve the encoder-decoder LSTM model for prediction individual VM CPU usage traces.

8.4 Conclusion and Future Directions

In this thesis, our focus was on improving scheduling high performance computing workloads in response to the recent increase in extensive computational processes. To do so, while keeping an eye on theoretical fundamentals, we tried to come up with solutions for real-world scenarios. Although there have been theoretical works to design algorithms for non-clairvoyance scheduling problems, the scheduling algorithms that use the knowledge about actual application resource usage patterns lead to considerable performance improvement.

We identified one of the challenges in applying the state of the art scheduling algorithms on clusters and the Cloud: the non-clairvoyance nature of the workloads. To overcome this challenge, we proposed a runtime reliability aware scheduling platform. Additionally, we proposed two approaches to predict application runtimes more accurately. We also studied the problem of predicting CPU usage of virtual machines and proposed the application of recurrent neural networks for this time series prediction problem.

In continuation of our work, several directions for workload prediction is apparent. More work on CPU usage of individual virtual machines with emerging open datasets from the public Cloud can be an interesting direction. Studying the application of emerging time series prediction approaches, including the Transformer (93), can be an interesting path for research. Predicting user submission patterns is another interesting time series prediction.

Bibliography

[azu] Microsoft azure public dataset. Accessed Oct. 25, 2012.

[wor] Parallel workloads archive. `http://cs.huji.ac.il/labs/parallel/workloads`, note = accessed: 2015-07-01.

[ec2] `http://aws.amazon.com/ec2/`.

[con] `http://www.vmware.com/consolidation/overview/`.

[5] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[6] Albagli-Kim, S., Shachnai, H., and Tamir, T. (2014). Scheduling jobs with dwindling resource requirements in clouds. In *INFOCOM, 2014 Proceedings IEEE*, pages 601–609. IEEE.

[7] Azmandian, F., Moffie, M., Dy, J. G., Aslam, J. A., and Kaeli, D. R. (2011). Workload characterization at the virtualization layer. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 63–72. IEEE.

[8] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[9] Bansal, N., Chan, H.-L., Khandekar, R., Pruhs, K., Schicber, B., and Stein, C. (2007). Non-preemptive min-sum scheduling with resource augmentation. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*, pages 614–624. IEEE.

[10] Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., and Schieber, B. (2001). A unified approach to approximating resource allocation and scheduling. *Journal of the ACM (JACM)*, 48(5):1069–1090.

[11] Bar-Shalom, Y., Li, X. R., and Kirubarajan, T. (2004). *Estimation with applications to tracking and navigation: theory algorithms and software*. John Wiley & Sons.

[12] Beloglazov, A., Abawajy, J., and Buyya, R. (2012). Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5):755–768.

[13] Bender, M. A., Muthukrishnan, S., and Rajaraman, R. (2004). Approximation algorithms for average stretch scheduling. *Journal of Scheduling*, 7(3):195–222.

[14] Bensusan, H. and Kalousis, A. (2001). Estimating the predictive accuracy of a classifier. In *European Conference on Machine Learning*, pages 25–36. Springer.

[15] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al. (2008). Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15.

[16] Bilmes, J. A. et al. (1998). A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. *International Computer Science Institute*, 4(510):126.

[17] Bishop, C. M. (1994). Mixture density networks. Technical report, Citeseer.

[18] Blazewicz, J., Kovalyov, M. Y., Machowiak, M., Trystram, D., and Weglarz, J. (2006). Preemptable malleable task scheduling problem. *IEEE Trans. Computers*, 55(4):486–490.

[19] Bobroff, N., Kochut, A., and Beaty, K. (2007). Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE.

[20] Bosnić, Z. and Kononenko, I. (2008a). Comparison of approaches for estimating reliability of individual regression predictions. *Data & Knowledge Engineering*, 67(3):504–516.

[21] Bosnić, Z. and Kononenko, I. (2008b). Estimation of individual prediction reliability using the local sensitivity analysis. *Applied intelligence*, 29(3):187–203.

[22] Brahma, P. P., Wu, D., and She, Y. (2015). Why deep learning works: A manifold disentanglement perspective. *IEEE transactions on neural networks and learning systems*, 27(10):1997–2008.

[23] Breitgand, D. and Epstein, A. (2012). Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds. In *INFOCOM, 2012 Proceedings IEEE*, pages 2861–2865. IEEE.

[24] Buyya, R. and Murshed, M. (2002). Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1175–1220.

[25] Caglar, F. and Gokhale, A. (2014). ioverbook: intelligent resource-overbooking to support soft real-time applications in the cloud. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 538–545. IEEE.

[26] Calzarossa, M. C., Massari, L., and Tessera, D. (2016). Workload characterization: A survey revisited. *ACM Computing Surveys (CSUR)*, 48(3):48.

[27] Chang, H., Kodialam, M., Kompella, R. R., Lakshman, T., Lee, M., and Mukherjee, S. (2011). Scheduling in mapreduce-like systems for fast completion time. In *INFOCOM, 2011 Proceedings IEEE*, pages 3074–3082. IEEE.

[28] Chekuri, C., Goel, A., Khanna, S., and Kumar, A. (2004). Multi-processor scheduling to minimize flow time with ε resource augmentation. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 363–372. ACM.

[29] Chekuri, C., Khanna, S., and Zhu, A. (2001). Algorithms for minimizing weighted flow time. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 84–93. ACM.

[30] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM.

[31] Chorowski, J. K., Bahdanau, D., Serdyuk, D., Cho, K., and Bengio, Y. (2015). Attention-based models for speech recognition. In *Advances in neural information processing systems*, pages 577–585.

[32] Cohn, T., Hoang, C. D. V., Vymolova, E., Yao, K., Dyer, C., and Haffari, G. (2016). Incorporating structural alignment biases into an attentional neural translation model. *arXiv preprint arXiv:1601.01085*.

[33] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. (2017). Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM.

[34] Dabbagh, M., Hamdaoui, B., Guizani, M., and Rayes, A. (2015a). Energy-efficient resource allocation and provisioning framework for cloud data centers. *IEEE Trans. Network and Service Management*, 12(3):377–391.

[35] Dabbagh, M., Hamdaoui, B., Guizani, M., and Rayes, A. (2015b). Toward energy-efficient cloud computing: Prediction, consolidation, and overcommitment. *IEEE network*, 29(2):56–61.

[36] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[37] Delimitrou, C. and Kozyrakis, C. (2014). Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144.

[38] Desai, N. (2005). Cobalt: an open source platform for hpc system software research. In *Edinburgh BG/L System Software Workshop*, pages 803–820.

[39] Di, S., Kondo, D., and Cirne, W. (2012). Characterization and comparison of cloud versus grid workloads. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 230–238. IEEE.

[40] Dimitriadou, S. and Karatza, H. (2010). Job scheduling in a distributed system using backfilling with inaccurate runtime computations. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 329–336. IEEE.

[41] Dobson, G. and Nambimadom, R. S. (2001). The batch loading and scheduling problem. *Operations research*, 49(1):52–65.

[42] Duggan, M., Mason, K., Duggan, J., Howley, E., and Barrett, E. (2017). Predicting host cpu utilization in cloud computing using recurrent neural networks. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 67–72. IEEE.

[43] Durbin, J. and Koopman, S. J. (2012). *Time series analysis by state space methods*, volume 38. OUP Oxford.

[44] Durillo, J. J. and Prodan, R. (2014). Multi-objective workflow scheduling in amazon ec2. *Cluster Computing*, 17(2):169–189.

[45] Duy, T. V. T., Sato, Y., and Inoguchi, Y. (2011). Improving accuracy of host load predictions on computational grids by artificial neural networks. *International Journal of Parallel, Emergent and Distributed Systems*, 26(4):275–290.

[46] Edmonds, J., Chinn, D. D., Brecht, T., and Deng, X. (2003). Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *J. Scheduling*, 6(3):231–250.

[47] Fan, Y., Rich, P., Allcock, W. E., Papka, M. E., and Lan, Z. (2017). Trade-off between prediction accuracy and underestimation rate in job runtime estimates. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 530–540. IEEE.

[48] Feitelson, D. G., Tsafrir, D., and Krakov, D. (2014). Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982.

[49] Fox, K. and Korupolu, M. (2013). Weighted flowtime on capacitated machines. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 129–143. SIAM.

[50] Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.

[51] Friedman, J. H. (2002). Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378.

[52] Gama, J. and Brazdil, P. (1995). Characterization of classification algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 189–200. Springer.

[53] Gammerman, A. and Vovk, V. (2007). Hedging predictions in machine learning. *The Computer Journal*, 50(2):151–163.

[54] Gao, P. X., Curtis, A. R., Wong, B., and Keshav, S. (2012). It's not easy being green. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 211–222. ACM.

[55] Gaussier, E., Glesser, D., Reis, V., and Trystram, D. (2015). Improving backfilling by using machine learning to predict running times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 64. ACM.

[56] Geer, D. (2007). For programmers, multicore chips mean multiple challenges. *Computer*, 40(9):17–19.

[57] Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm.

[58] Gibbons, R. (1997). A historical application profiler for use by parallel schedulers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 58–77. Springer.

[59] Gong, Z., Gu, X., and Wilkes, J. (2010). Press: Predictive elastic resource scaling for cloud systems. *CNSM*, 10:9–16.

[60] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[61] Graves, A. and Jaitly, N. (2014). Towards end-to-end speech recognition with recurrent neural networks. In *International Conference on Machine Learning*, pages 1764–1772.

[62] Graves, A., Jaitly, N., and Mohamed, A.-r. (2013). Hybrid speech recognition with deep bidirectional lstm. In *2013 IEEE workshop on automatic speech recognition and understanding*, pages 273–278. IEEE.

[63] Graves, A. and Schmidhuber, J. (2009). Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552.

[64] Greenberg, A., Hamilton, J., Maltz, D. A., and Patel, P. (2008). The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73.

[65] Gupta, S. and Dinesh, D. A. (2017). Resource usage prediction of cloud workloads using deep bidirectional long short term memory networks. In *2017 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, pages 1–6. IEEE.

[66] Hastie, T., Tibshirani, R., and Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition.* Springer series in statistics. Springer.

[67] Hazan, E. et al. (2016). Introduction to online convex optimization. *Foundations and Trends® in Optimization*, 2(3-4):157–325.

[68] Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier.

[69] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[70] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[71] Hovestadt, M., Kao, O., Keller, A., and Streit, A. (2003). Scheduling in hpc resource management systems: Queuing vs. planning. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–20. Springer.

[Im et al.] Im, S., Naghshnejad, M., and Singhal, M. Scheduling jobs with non-uniform demands on multiple servers without interruption.

[73] Ismaeel, S. and Miri, A. (2015). Using elm techniques to predict data centre vm requests. In *Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on*, pages 80–86. IEEE.

[74] Itti, L., Koch, C., and Niebur, E. (1998). A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on pattern analysis and machine intelligence*, 20(11):1254–1259.

[75] Jansen, K. and Zhang, G. (2007). Maximizing the total profit of rectangles packed into a rectangle. *Algorithmica*, 47(3):323–342.

[76] Jennings, B. and Stadler, R. (2015). Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23(3):567–619.

[77] JoSEP, A. D., KAtz, R., KonWinSKi, A., Gunho, L., PAttERSon, D., and RABKin, A. (2010). A view of cloud computing. *Communications of the ACM*, 53(4).

[78] Kc, K. and Anyanwu, K. (2010). Scheduling hadoop jobs to meet deadlines. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 388–392. IEEE.

[79] Kendall, A. and Gal, Y. (2017). What uncertainties do we need in bayesian deep learning for computer vision? In *Advances in neural information processing systems*, pages 5574–5584.

[80] Khan, A., Yan, X., Tao, S., and Anerousis, N. (2012). Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294. IEEE.

[81] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[82] Klusáček, D., Chlumskỳ, V., and Rudová, H. (2015). Planning and optimization in torque resource manager. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 203–206. ACM.

[83] Klusáček, D. and Rudová, H. (2010). Alea 2: job scheduling simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 61. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[84] Koh*, S.-G., Koo, P.-H., Ha, J.-W., and Lee, W.-S. (2004). Scheduling parallel batch processing machines with arbitrary job sizes and incompatible job families. *International Journal of Production Research*, 42(19):4091–4107.

[85] Kong, F. and Liu, X. (2015). A survey on green-energy-aware power management for datacenters. *ACM Computing Surveys (CSUR)*, 47(2):30.

[86] Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76.

[87] Lakshminarayanan, B., Pritzel, A., and Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems*, pages 6402–6413.

[88] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

[89] Lee, C. B., Schwartzman, Y., Hardy, J., and Snavely, A. (2004). Are user runtime estimates inherently inaccurate? In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 253–263. Springer.

[90] Lee, Y. C., Han, H., Zomaya, A. Y., and Yousif, M. (2015). Resource-efficient workflow scheduling in clouds. *Knowledge-Based Systems*, 80:153–162.

[91] Li, F., Cao, J., Wang, X., Sun, Y., and Sahni, Y. (2017). Enabling software defined networking with qos guarantee for cloud applications. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 130–137. IEEE.

[92] Li, G., Ma, Q., Tang, H., Paterson, A. H., and Xu, Y. (2009). Qubic: a qualitative biclustering algorithm for analyses of gene expression data. *Nucleic acids research*, 37(15):e101–e101.

[93] Li, S., Jin, X., Xuan, Y., Zhou, X., Chen, W., Wang, Y.-X., and Yan, X. (2019). Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *arXiv preprint arXiv:1907.00235*.

[94] Liang, D., Ho, P.-J., and Liu, B. (2000). Scheduling in distributed systems. *Department of Computer Science and Engineering University of California, San Diego*.

[95] Lin, W., Peng, B., Liang, C., and Liu, B. (2013). Novel resource allocation model and algorithms for cloud computing. In *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*, pages 77–82. IEEE.

[96] Lipton, Z. C., Berkowitz, J., and Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

[97] Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

[98] Maguluri, S. T. and Srikant, R. (2014). Scheduling jobs with unknown duration in clouds. *Networking, IEEE/ACM Transactions on*, 22(6):1938–1951.

[99] Maguluri, S. T., Srikant, R., and Ying, L. (2014). Heavy traffic optimal resource allocation algorithms for cloud computing clusters. *Performance Evaluation*, 81:20–39.

[100] Matsunaga, A. and Fortes, J. A. (2010). On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE Computer Society.

[101] McLachlan, G. J. and Basford, K. E. (1988). *Mixture models: Inference and applications to clustering*, volume 84. M. Dekker New York.

[102] Mendes, C. L. and Reed, D. A. (1998). Integrated compilation and scalability analysis for parallel systems. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 385–392. IEEE.

[103] Mishra, A. K., Hellerstein, J. L., Cirne, W., and Das, C. R. (2010). Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41.

[104] Mu'alem, A. W. and Feitelson, D. G. (2001). Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543.

[105] Nguyen, H., Shen, Z., Gu, X., Subbiah, S., and Wilkes, J. (2013). Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, volume 13, pages 69–82.

[106] Nguyen, T. H., Di Francesco, M., and Yla-Jaaski, A. (2017). Virtual machine consolidation with multiple usage prediction for energy-efficient cloud data centers. *IEEE Transactions on Services Computing*.

[107] Nielsen, M. A. (2015). *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA:.

[108] Niranjan Mysore, R., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., and Vahdat, A. (2009). Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 39–50.

[109] Nissimov, A. and Feitelson, D. G. (2007). Probabilistic backfilling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 102–115. Springer.

[110] Nouretdinov, I., Melluish, T., and Vovk, V. (2001). Ridge regression confidence machine. In *ICML*, pages 385–392.

[111] Novakovic, D., Vasic, N., Novakovic, S., Kostic, D., and Bianchini, R. (2013). Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Annual Technical Conference*, number EPFL-CONF-185984.

[112] Ophem, S. v. and Berkhoff, A. P. (2016). A numerically stable, finite memory, fast array recursive least squares filter for broadband active noise control. *International journal of adaptive control and signal processing*, 30(1):31–45.

[113] Paroli, R., Spezia, L., et al. (2002). Parameter estimation of gaussian hidden markov models when missing observations occur. *Metron-International Journal of Statistics*, 60(3-4):163–179.

[114] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318.

[115] Pinedo, M. L. (2012). *Scheduling: theory, algorithms, and systems*. Springer Science & Business Media.

[116] Qin, Y., Song, D., Chen, H., Cheng, W., Jiang, G., and Cottrell, G. (2017). A dual-stage attention-based recurrent neural network for time series prediction. *arXiv preprint arXiv:1704.02971*.

[117] Ramírez-Velarde, R., Tchernykh, A., Barba-Jimenez, C., Hirales-Carbajal, A., and Nolazco-Flores, J. (2017). Adaptive resource allocation with job runtime uncertainty. *Journal of Grid Computing*, 15(4):415–434.

[118] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. (2012). Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM.

[119] Rodrigues, P. P., Gama, J., and Bosnic, Z. (2008). Online reliability estimates for individual predictions in data streams. In *2008 IEEE International Conference on Data Mining Workshops*, pages 36–45. IEEE.

[120] Ross, S., Mineiro, P., and Langford, J. (2013). Normalized online learning. *arXiv preprint arXiv:1305.6646*.

[121] Sanders, P. and Speck, J. (2012). Energy efficient frequency scaling and scheduling for malleable tasks. In *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, pages 167–178.

[122] Sandholm, T. and Lai, K. (2010). Dynamic proportional share scheduling in hadoop. In *Job scheduling strategies for parallel processing*, pages 110–131. Springer.

[123] Saunders, C., Gammerman, A., and Vovk, V. (1999). Transduction with confidence and credibility.

[124] Schäfer, A. M. and Zimmermann, H.-G. (2007). Recurrent neural networks are universal approximators. *International journal of neural systems*, 17(04):253–263.

[125] Schopf, J. M. and Berman, F. (1999). Using stochastic intervals to predict application behavior on contended resources. In *Parallel Architectures, Algorithms,*

and Networks, 1999.(I-SPAN'99) Proceedings. Fourth InternationalSymposium on, pages 344–349. IEEE.

[126] Sharma, B., Chudnovsky, V., Hellerstein, J. L., Rifaat, R., and Das, C. R. (2011). Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 3. ACM.

[127] Sharma, S., Kiros, R., and Salakhutdinov, R. (2015). Action recognition using visual attention. *arXiv preprint arXiv:1511.04119*.

[128] Skovira, J., Chan, W., Zhou, H., and Lifka, D. (1996). The easy loadleveler api project. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47. Springer.

[129] Smith, W., Foster, I., and Taylor, V. (1998). Predicting application run times using historical information. In Feitelson, D. G. and Rudolph, L., editors, *Job Scheduling Strategies for Parallel Processing*, pages 122–142, Berlin, Heidelberg. Springer Berlin Heidelberg.

[130] Smith, W. E. (1956). Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66.

[131] Song, B., Yu, Y., Zhou, Y., Wang, Z., and Du, S. (2018). Host load prediction with long short-term memory in cloud computing. *The Journal of Supercomputing*, 74(12).

[132] Song, W., Xiao, Z., Chen, Q., and Luo, H. (2014). Adaptive resource provisioning for the cloud using online bin packing. *Computers, IEEE Transactions on*, 63(11):2647–2660.

[133] Sonmez, O., Yigitbasi, N., Iosup, A., and Epema, D. (2009). Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 111–120. ACM.

[134] Srikantaiah, S., Kansal, A., and Zhao, F. (2008). Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems*, volume 10, pages 1–5. San Diego, California.

[135] Srinivasan, S., Kettimuthu, R., Subramani, V., and Sadayappan, P. (2002). Characterization of backfilling strategies for parallel job scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 514–519. IEEE.

[136] Steinberg, A. (1997). A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409.

[137] Stillwell, M., Vivien, F., and Casanova, H. (2010). Dynamic fractional resource scheduling for hpc workloads. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE.

[138] Talby, D. and Feitelson, D. G. (1999). Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 513–517. IEEE.

[139] Tang, W., Desai, N., Buettner, D., and Lan, Z. (2010). Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE.

[140] Tang, W., Lan, Z., Desai, N., and Buettner, D. (2009). Fault-aware, utility-based job scheduling on blue, gene/p systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE.

[141] Tchernykh, A., Schwiegelsohn, U., Alexandrov, V., and Talbi, E.-g. (2015). Towards understanding uncertainty in cloud computing resource provisioning. *Procedia Computer Science*, 51:1772–1781.

[142] Tsafrir, D., Etsion, Y., and Feitelson, D. G. (2007). Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803.

[143] Tuda, K., Rätsch, G., Mika, S., and Müller, K.-R. (2001). Learning to predict the leave-one-out error of kernel based classifiers. In *International Conference on Artificial Neural Networks*, pages 331–338. Springer.

[144] Uzsoy, R. (1994). Scheduling a single batch processing machine with non-identical job sizes. *THE INTERNATIONAL JOURNAL OF PRODUCTION RESEARCH*, 32(7):1615–1635.

[145] Vanschoren, J. (2019). Meta-learning. In *Automatic Machine Learning: Methods, Systems, Challenges*, pages 39–68. Springer.

[146] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

[147] Wang, M., Meng, X., and Zhang, L. (2011). Consolidating virtual machines with dynamic bandwidth demand in data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 71–75. IEEE.

[148] Wang, W., Li, B., and Liang, B. (2014). Dominant resource fairness in cloud computing systems with heterogeneous servers. In *INFOCOM, 2014 Proceedings IEEE*, pages 583–591. IEEE.

[149] Weigend, A. S. and Nix, D. A. (1994). Predictions with confidence intervals (local error bars). In *Proceedings of the international conference on neural information processing*, pages 847–852.

[150] Williamson, D. P. and Shmoys, D. B. (2011). *The Design of Approximation Algorithms*. Cambridge University Press.

[151] Wolski, R. and Brevik, J. (2013). Using parametric models to represent private cloud workloads. *IEEE Transactions on Services Computing*, 7(4):714–725.

[152] Xhafa, F., Carretero, J., Dorronsoro, B., and Alba, E. (2012). A tabu search algorithm for scheduling independent jobs in computational grids. *Computing and informatics*, 28(2):237–250.

[153] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057.

[154] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer.

[155] You, Q., Jin, H., Wang, Z., Fang, C., and Luo, J. (2016). Image captioning with semantic attention. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4651–4659.

[156] Yuan, H., Bi, J., Zhang, J., Tan, W., and Huang, K. (2017). Workload-aware revenue maximization in sdn-enabled data center. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 18–25. IEEE.

[157] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10:10–10.

[158] Zhang, Q., Hellerstein, J. L., and Boutaba, R. (2011). Characterizing task usage shapes in google's compute clusters. In *Large Scale Distributed Systems and Middleware Workshop (LADIS'11)*.

[159] Zhang, Q., Zhani, M. F., Zhang, S., Zhu, Q., Boutaba, R., and Hellerstein, J. L. (2012). Dynamic energy-aware capacity provisioning for cloud computing environments. In *Proceedings of the 9th international conference on Autonomic computing*, pages 145–154. ACM.

[160] Zheng, X., Zhou, Z., Yang, X., Lan, Z., and Wang, J. (2016). Exploring plan-based scheduling for large-scale computing systems. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 259–268. IEEE.

[161] Zheng, Y., Shroff, N. B., and Sinha, P. (2013). A new analytical technique for designing provably efficient mapreduce schedulers. In *INFOCOM, 2013 Proceedings IEEE*, pages 1600–1608. IEEE.