Abstract of “Application-Aware Cluster Resource Management” by Jeffrey Thomas Rasley, Ph.D., Brown University, April 2019.

The adoption of cloud computing has pushed many big-data analytics frameworks to run as multi-tenant services where users submit jobs and a cluster resource manager is responsible for provisioning resources and executing jobs. In order to support a wide-range of data processing models, designers of modern cluster resource management frameworks advocate for a strict decoupling of resource management infrastructure (e.g., Apache YARN) from specific data processing models (e.g., Hadoop, TensorFlow, Spark).

This dissertation introduces application-aware resource management, which advocates for judiciously pushing semantics from the application into the cluster manager in order for it to make more informed scheduling decisions. We show that application-aware cluster management can lead to significant gains in terms of decreased job completion times, increased cluster utilization, and improved job utility. We present prototypes that exemplify the benefits of our approach in systems we call Yaq and HyperDrive.

We identify inefficiencies in existing cluster management framework designs that lead to poor utilization and degraded user-experience in the presence of heterogeneous workloads. We present Yaq as a cluster manager which is able to prioritize task execution based on application-level metrics such as task durations or total amount of work remaining in a job. We show that Yaq can significantly improve both cluster utilization and workload completion times.

We identify several difficulties and inefficiencies in the building and training of machine-learning models in the context of shared clusters. We present HyperDrive as a cluster manager which is able to classify and prioritize jobs based on user-defined utility (e.g., model accuracy). We implement a scheduling algorithm in HyperDrive that uses probabilistic model-based classification with dynamic scheduling and early termination to improve model performance and decrease workload completion times.

Application-Aware Cluster Resource Management

by

Jeffrey Thomas Rasley

B. S. E., University of Washington, 2012

M. Sc., Brown University, 2014

A dissertation submitted in partial fulfillment of the

requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science at Brown University

Providence, Rhode Island

April 2019

© Copyright 2019 by Jeffrey Thomas Rasley

This dissertation by Jeffrey Thomas Rasley is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ______________________________

Rodrigo Fonseca, Director

Recommended to the Graduate Council

Date ______________ ______________________________

Yuxiong He, Reader
Microsoft

Date ______________ ______________________________

Tim Kraska, Reader
Massachusetts Institute of Technology

Approved by the Graduate Council

Date ______________ ______________________________

Andrew G. Campbell
Dean of the Graduate School

Acknowledgements

This work was generously supported by the National Science Foundation (DGE-1058262) and Brown University. I am very grateful to the NSF for funding my research, it gave me the freedom to pursue any and all research directions I found interesting along my journey. In addition, none of this dissertation would have been possible without the unwavering support of my adviser Rodrigo Fonseca. Rodrigo's support and encouragement throughout my PhD was a true inspiration. I would also like to thank my committee members Yuxiong He and Tim Kraska.

I am incredibly thankful for three different Microsoft Research internships and contracting positions that I had during my PhD. These opportunities allowed me to collaborate and learn from a truly amazing set of researchers. Specifically, Yuxiong He and Konstantinos Karanasos had a significant and positive impact on my research. In addition, the wonderful support of Srikanth Kandula, Olatunji Ruwase, Sriram Rao, and Trishul Chilimbi cannot be overlooked.

Early on in my PhD I had the truly wonderful opportunity of interning at IBM Research in Austin, TX. Colin Dixon's support and encouragement cannot be understated, I hope to help a PhD student half as much as Colin has helped me through the years. In addition, working with Eric Rozner, Brent Stephens, Wes Felter, John Carter, and Kanak Agarwal was amazing. Also, reflecting about my time at IBM Research and not thanking the excellent Austin barbecue and craft beer scene would not be right.

I have to thank Justin Cappos, who was my undergraduate research adviser at the University of Washington. Justin introduced me to research and encouraged me to pursue a PhD. If I hadn't met Justin at the undergraduate research fair this document may not have been possible.

The support of the systems lab at Brown cannot be understated. Early on the support and

encouragement of Andrew Ferguson was instrumental in my success. In the latter part of my PhD it was support and encouragement of Jonathan Mace. A special thanks to the rest of my labmates who made our lab a great place to call home: Marcelo Martins, Da Yu, Ray Zhou, Michael Markovitch, Nicholas DeMarinis, Linnan Wang, Usama Naseer, Saim Salman, Sumukha Tumkur Vani, George Hongkai Sun, and Ryan Roelke.

I am thankful for all of the friendships I made during my time at Brown. My PhD experience was greatly improved by your friendships. Thank you: Connor Gramazio, John Meehan, Chris Tanner, Kavosh Asadi, Hannah Quay-de la Vallee, Betsy Hilliard, Tim Nelson, Kayhan Dursun, and many more.

I am especially thankful for the friendship of Kyle Tracy and Kerani McClelland. They were always there to help encourage me and share beers with, which made our time together in Rhode Island feel much closer to home. In addition, thank you to our west-coast friends who made frequent trips over the years to visit us: Aaron Danneker, Daniel Powell, Allison Obourn, and Danielle Shine.

Finally, literally none of this would be possible without the unconditional support of my family. My wife Danielle's constant love and support helped to uplift my spirits during the low-points and amplify the high-points. I couldn't imagine taking this journey with anyone else, thank you for everything. I also thank Eleanor, my amazing daughter, who gave me a new inspiration to finish strong. In addition, I could not have done any of this without my parents unconditional love and support throughout my entire life. I feel incredibly lucky to have had them as parents. Thank you for all your support even though it meant being almost three thousand miles apart for so many years. Thank you for visiting us as often as you did, it made the distance feel significantly smaller.

Contents

List of Figures x

1 Introduction 1

1.1 Application-aware Cluster Resource Management 1

1.2 Queue Management for Data Analytics Clusters 2

1.3 Utility-based Cluster Management 3

1.4 Summary of Contributions 4

2 Yaq: Application-Aware Queue Management 6

2.1 Motivation 6

2.2 Requirements 9

2.3 Task Queuing Overview 10

2.4 Basic System Components 11

2.5 Centralized Scheduling With Queues (Yaq-c) 12

2.6 Distributed Scheduling With Queues (Yaq-d) 13

3 Worker Node and Global Queue Management 14

3.1 Bounding Queue Lengths 15

3.1.1 Length-Based Queue Bounding 15

3.1.2 Determining the Queue Length 17

3.1.3 Delay-Based Queue Bounding 22

3.2 Placement of Tasks to Nodes 22

3.3 Prioritizing Task Execution 25
3.4 Global Queue Management Policies 28
3.4.1 Global Job Reordering 28
3.4.2 Sharing Policies 29

4 Evaluation of Yaq **31**
4.1 Implementation 31
4.2 Experimental Evaluation 32
4.3 Experimental Setup 33
4.4 Evaluating Yaq-c 34
4.5 Evaluating Yaq-d 35
4.6 Imposing Sharing Constraints 37
4.7 Micro-experiments 37
4.7.1 Bounding Queuing Delays 38
4.7.2 Task Placement at Queues 39
4.7.3 Task Prioritization at Worker Nodes 40
4.7.4 Heavy-tailed Workload 42
4.8 Related Work 43

5 HyperDrive: Utility-Based Cluster Management **45**
5.1 Motivation 45

6 Utility-Based Cluster Scheduling Design Principles **49**
6.1 Identify Poor Configurations Early 50
6.2 Classify Promising Configurations Early and Judiciously 51
6.3 Resource Allocation Between Promising and Opportunistic Configurations 54
6.4 Design Overview of HyperDrive 54
6.4.1 Design Considerations 54
6.4.2 HyperDrive Framework 55

7 Scheduling Algorithm and Framework Implementation **59**

7.1 Configuration Classification 59

7.2 Infused Classification & Scheduling Methodology 63

7.3 Implementation of HyperDrive 66

7.4 Suspend & Resume Support 66

7.5 Learning Curve Prediction 67

7.6 Scheduling Policies 68

8 Evaluation of HyperDrive **70**

8.1 Experimental Setup 70

8.2 Supervised-Learning 72

8.2.1 Job Execution Duration 72

8.2.2 Scheduling Performance Comparison 73

8.2.3 Scheduling Overhead 74

8.3 Reinforcement-Learning 75

8.3.1 Scheduling Performance Comparison 76

8.3.2 Scheduling Overhead 77

8.4 HyperDrive Sensitivity Analysis 78

8.4.1 Simulator 78

8.4.2 Supervised-Learning 79

8.4.3 Reinforcement-Learning 82

9 Discussion of HyperDrive Considerations **84**

9.1 Discussion 84

9.2 Related Work 86

10 Conclusion **88**

10.1 Big-Data Analytics Systems 88

10.2 Machine-Learning Model Optimization Systems 89

10.3 Future Work 89
10.3.1 Application-Aware Load Balancing 89
10.3.2 Multi-Metric Machine-Learning Optimization Systems 90

List of Figures

2.1 Job and task durations for two production workloads. 7

2.2 Job completion times for production Workload 2 using different scheduling types. . 9

2.3 System architecture for centralized scheduling (Yaq-c). 12

2.4 System architecture for distributed scheduling (Yaq-d). 13

3.1 Queue length required to achieve cluster utilization $u{=}1{-}\delta$, given the number of tasks that can be processed by a node per heartbeat interval ($r\mu\tau$). 16

4.1 Job completion times for Yaq-c on Hive-MS workload. 34

4.2 Job completion times for Yaq-d on Hive-MS workload. 35

4.3 Imposing capacity constraints in Yaq-c. 37

4.4 Average cluster slot utilization with different workloads and queue lengths. 38

4.5 Job completion time for GridMix-MS workload with different queue bounding techniques and no task prioritization. 38

4.6 Job completion time for the GridMix-MS workload with different RM task placement policies. 40

4.7 Job completion time for GridMix-MS workload with different task prioritization algorithms. 40

4.8 Job completion time for the GridMix-MS workload with different hard starvation thresholds (ST) and STF prioritization. 41

4.9 Job completion time for the heavy-tailed workload that is based on GridMix-MS. . . 42

5.1 Performance of 50 randomly selected supervised-learning hyperparameter configurations. 46

6.1 Final validation accuracy distribution of 90 randomly selected CIFAR-10 configurations. 50
6.2 Final learning curves for two different configurations A and B. 51
6.3 Job A has higher predicted accuracy but lower confidence, while Job B has better final accuracy and higher confidence. 52
6.4 Validation accuracy predictions at epoch 10. 53
6.5 Validation accuracy predictions at epoch 30. 53
6.6 Final validation accuracy at epoch 120. 53
6.7 HyperDrive architecture 55

7.1 Desired slots are low early on due to low confidence. 65
7.2 Desired slots are high later on due to higher confidence. 65
7.3 Ratio of promising slots increases over time. 66

8.1 Job execution duration distribution comparing different scheduling policies with supervised-learning workload. 72
8.2 Time to reach target validation accuracy (CIFAR-10). 73
8.3 Suspend latency for supervised-learning workload 74
8.4 Model snapshot filesize for supervised-learning workload. 75
8.5 Performance of 15 randomly selected LunarLander model configurations over 20,000 episode trials. 75
8.6 Time to reach target reward (LunarLander). 76
8.7 Suspend latency distribution for LunarLander workload. 77
8.8 Suspend snapshot size distribution for LunarLander workload. 78
8.9 Simulator Design. 79
8.10 Simulator validation (LunarLander). 80
8.11 Resource capacities (CIFAR-10). 80

8.12 Random job orderings (CIFAR-10). 81
8.13 Time to reach target reward with LunarLander using different scheduling policies and resource capacities. 82
8.14 Time to reach target reward for LunarLander with 25 random configuration orders using different policies. 83

Chapter 1

Introduction

Thesis Statement Cluster schedulers can benefit from a judiciously widened interface between resource management and application boundaries. Using this wider interface to exchange application-level semantics between applications and cluster schedulers can significantly improve performance and cluster utility.

1.1 Application-aware Cluster Resource Management

The adoption of cloud computing has pushed many big data analytics frameworks to run as multi-tenant services where users can submit jobs and a resource manager is responsible for provisioning resources and executing jobs. In order to support a wide-range of data processing models designers of modern cluster resource management frameworks advocate for a strict decoupling of resource management infrastructure from specific data processing models (e.g., Hadoop, TensorFlow, Spark). In this thesis, we present *application-aware* resource scheduling which instead advocates for a loosening of this abstraction boundary, in a principled way, in order for resource management frameworks to learn more about the metrics that are important to the application's they are scheduling. By loosening this abstraction boundary we show, through several example scenarios, that job completion times decrease, overall cluster utilization increases, and cluster operators and users receive a higher return on their costly investment.

Typical resource management frameworks are agnostic to the actual jobs running on them. This design is typically useful so that there is a decoupling of the resource management infrastructure and the actual programming model being used in the system (e.g., Hadoop, TensorFlow, Spark, Tez). A popular example of this decoupling can be seen in the YARN framework [59], which was built specifically to address design constraints in the original Apache Hadoop framework that had a strict coupling of resource management and the MapReduce programming model. In this thesis we show the benefits of application-aware resource scheduling through two examples, section 1.2 and section 1.3.

1.2 Queue Management for Data Analytics Clusters

In recent years, organizations small and large have been taking advantage of large scale-out commodity clusters to help with big data analytics problems with the help of frameworks like Hadoop [28] and Spark [67]. These clusters run a wide variety of applications including batch data analytics jobs, machine learning jobs, and interactive queries. To reduce operational costs, and, therefore, improve return on investment, there is a trend toward consolidating diverse workloads onto shared clusters. However, doing so places strain on the cluster scheduler due to the diversity of job requirements needed by different workloads.

Cluster schedulers such as YARN [59] and Borg [60] use a logically centralized resource manager (RM) which is responsible for managing cluster resources by specifically matching potential jobs with the resources they need to execute on worker machines. Typically this architecture involves worker machines exchanging heartbeat messages with the RM on the order of seconds. Heartbeats are used to report resource availability to the RM so that it can contain a global view of available resources that it can schedule. This design has two primary problems: (1) the RM is in the critical path of all scheduling decisions. (2) whenever a task finishes resources can remain idle between heartbeats. When workloads consist of a mix of long and short task durations this heartbeat-based architecture can negatively affect cluster utilization. Even worse, jobs that consist of short tasks can suffer from lengthened job completion times.

Distributed scheduler alternatives have been proposed that avoid logical centralization. Apollo [13], Sparrow [44], and others allow job managers to independently decide where to execute their tasks, either to improve scalability or to reduce allocation latency. In these systems workloads that consist of short-lived tasks become less of a problem, because tasks can be scheduled onto queues at worker machines by each job manager. However, the lack of a global cluster view causes these designs to be vulnerable to other problems. Each scheduler makes a locally optimum allocation which may cause scheduling collisions at worker nodes. Task assignments are vulnerable to head-of-line blocking when tasks are mixed with long and short task durations. We observed that these aspects affect job completion times, leading to increased tail latency and unpredictability in job run times.

This thesis introduces a collection of queue management techniques we collectively refer to as Yaq [1]. In this work we studied job heterogeneity in production clusters at Microsoft. We found that task durations vary from a few milliseconds to tens of thousands of seconds. Moreover, a significant fraction of tasks are short-lived ($\sim$50% last less than 10s), which illustrates a generally observed shift towards smaller tasks [35, 43, 44]. In order to mitigate problems that arise with heterogeneous workloads we introduce queue management techniques that infuse application-level information into worker nodes to allow for dynamic local scheduling of tasks.

1.3 Utility-based Cluster Management

In recent years, organizations with large scale-out commodity clusters have been shifting the type of big-data analytics they perform to leverage the latest techniques in machine-learning (ML). This type of analysis typically involves a data scientist designing, building, and training statistical models (e.g., neural networks) that leverage the large amounts of data at the organizations disposal. The building and training of these statistical models is typically executed on a shared cluster using a resource manager such as YARN with one of the many popular machine-learning frameworks at its core [2, 34, 65].

The task of building a high performing and successful model is heavily reliant on the selection

[1]This work was previously published [46], its content has been revised and included throughout this dissertation.

of many different adjustable parameters, commonly called hyperparameters, which are configured prior to training a model. Examples of these hyperparameters include learning rate (in many models), number and size of hidden layers in a deep neural network, number of clusters in k-means clustering, and many more. It is often difficult for data scientists to properly select hyperparameter values that produce desired results. This difficulty often leads data scientists to automatic or manual hyperparameter exploration to search across different configurations of a model, where each configuration represents a specific set of hyperparameter values. The goal of this search is to find configurations that optimize the model's utility (e.g., high predictive accuracy, low loss, high reward) with affordable time and cost. Hyperparameter exploration involves two related problems: generating candidate configurations from the large space of hyperparameter values, and efficiently scheduling these configurations in a shared cluster.

This thesis introduces an application-aware cluster scheduling algorithm (POP) based on model utility and a cluster scheduling framework (HyperDrive) for hyperparameter exploration [2]. Our POP scheduling algorithm dynamically classifies running configurations into categories based on observed application-level utility metrics (e.g., model accuracy). Furthermore, POP infuses probabilistic model-based classification with dynamic scheduling and early termination to jointly optimize utility and cost of a configuration. In addition, we build a comprehensive hyperparameter exploration infrastructure, HyperDrive, to support existing and future scheduling algorithms for a wide range of usage scenarios across different machine-learning frameworks and learning domains.

1.4 Summary of Contributions

In summary, we make the following contributions in this thesis:

- We introduce the concept of application-aware cluster management and apply it across two domains: big-data analytics systems and machine-learning model optimization systems.
- We identify inefficiencies in existing cluster management framework designs in the presence of heterogeneous workloads that lead to poor utilization and degraded user-experience. We

[2] This work was previously published [45], its content has been revised and included throughout this dissertation.

show that through the addition of worker-side queuing, careful sharing of application-level metrics along with queue management we can mitigate these inefficiencies and improve overall cluster user experience.

- We present two implementations of our queue management techniques across two different cluster scheduling designs: a centralized scheduler (Yaq-c) and a distributed scheduler (Yaq-d). We show that in both designs we can significantly improve both cluster utilization and workload completion latency [46].

- We identify several difficulties and inefficiencies in the building and training machine-learning models in the context of shared clusters. We design and implement an application-aware cluster scheduling algorithm that mitigates these challenges through the use of probabilistic model-based configuration classification with dynamic scheduling and early termination.

- We present a cluster scheduling implementation, HyperDrive, that that not only facilitates our scheduling algorithm, but supports existing and future scheduling algorithms, and works with different machine-learning domains and frameworks [45].

Chapter 2

Yaq: Application-Aware Queue Management

2.1 Motivation

Data-parallel frameworks [48, 59, 67] and scale-out commodity clusters are being increasingly used to store and extract value from data. While some enterprises have large clusters, many others use public cloud providers. Such clusters run a wide variety of applications including batch data analytics jobs, machine learning jobs and interactive queries. To reduce operational costs, and, therefore, improve return on investment, there is a trend toward consolidating diverse workloads onto shared clusters. However, doing so places considerable strain in the cluster scheduler, which has to deal with vastly heterogeneous jobs, while maintaining high cluster utilization, fast and predictable job completion times, and offering expressive sharing policies among users.

To showcase the job heterogeneity in production clusters, we provide task and job durations for two production workloads of Microsoft (see Figure 2.1), we refer to these as Workload 1 and Workload 2.[1] Task durations vary from a few milliseconds to tens of thousands of seconds. Moreover, a significant fraction of tasks are short-lived ($\sim$50% last less than 10s), which illustrates a generally observed shift towards smaller tasks [35, 43, 44].

[1] Durations only for successful (non-failed) tasks are included in the figure.

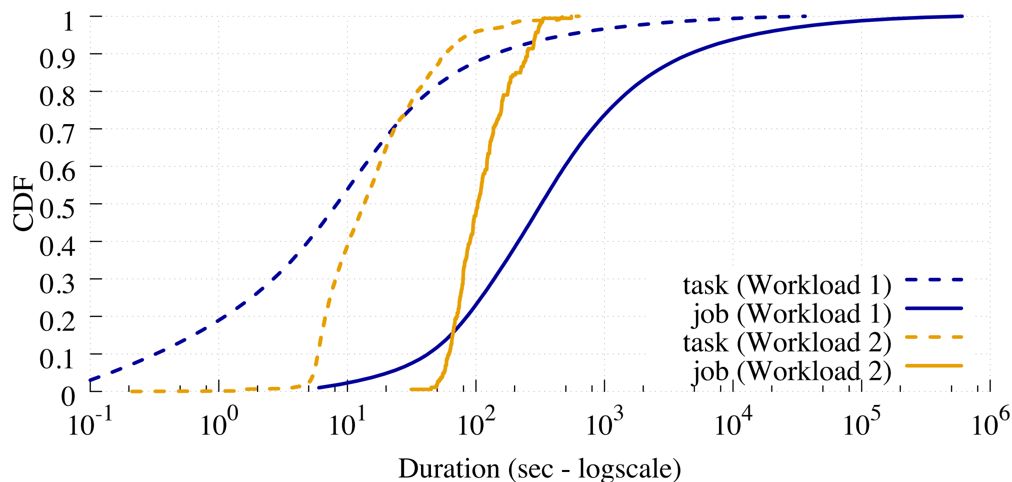

Figure 2.1: Job and task durations for two production workloads.

Cluster schedulers such as YARN [59] and Borg [60] have a logically centralized service, often called the resource manager (RM), which serves as a matchmaker between the resource needs of various jobs and the available resources on worker machines. Typically, machines exchange heartbeat messages with the RM once every few seconds,[2] and are initiated either by worker machines (in YARN) or by the RM (in Borg). Through heartbeats, worker machines report resource availability to the RM, which in turn determines the allocation of tasks to machines. This design has two main problems: first, the RM is in the critical path of all scheduling decisions; second, whenever a task finishes, resources can remain fallow between heartbeats. These aspects slow down job completion: a job with a handful of short tasks can take tens of seconds to finish. Worse, they affect cluster utilization especially when tasks are short-lived. Table 2.1 shows the average cluster utilization (i.e., the percentage of occupied slots) with tasks of different durations for an 80-node YARN cluster. The label X sec denotes a synthetic workload wherein every task lasts X seconds. The label Mixed-5-50 is an even mix of 5 and 50 sec tasks. Workload 1 is the production workload shown in Figure 2.1. We see that as task durations get shorter, cluster utilization drastically degrades, and can be as low as 61%.

[2] YARN clusters with ∼4K nodes use a heartbeat interval of 3sec [59]; the Borg RM polls each machine every few secs with the 95th percentile being below 10sec in a ∼10K-node cluster [60].

5 sec	10 sec	50 sec	Mixed-5-50	Workload 1
60.59%	78.35%	92.38%	78.54%	83.38%

Table 2.1: Average YARN cluster slot utilization for workloads with varying task durations.

A few schedulers avoid logical centralization. Apollo [13], Sparrow [44] and others allow job managers to independently decide where to execute their tasks, either to improve scalability (in terms of cluster size or scheduling decisions rate) or to reduce allocation latency. The above problem with short-lived tasks becomes less prevalent, because tasks can be *pushed* onto queues at worker machines by each job manager. However, these schedulers are vulnerable to other problems: (a) each job manager achieves a *local* optimum allocation, but coordination across various job managers to achieve *globally* optimal allocations is not possible;[3] (b) worse, the distributed schedulers do not always pick appropriate machines since they fail to account for the pending work in each queue; (c) the assignments are vulnerable to head-of-line blocking when tasks have heterogeneous resource demands and durations. These aspects affect job completion times, leading to increased tail latency and unpredictability in job run times.

To illustrate these aspects, Figure 2.2 presents a CDF of job completion times for Workload 2 with YARN and an implementation of Sparrow's batch sampling on Mercury [35, 44]. We see that the latter improves some very short jobs, but has a long tail of jobs that exhibit longer completion times. As we will see later, this happens because batch sampling fails to make globally optimal task placement decisions, and because FIFO queues at worker nodes suffer from head-of-line blocking. Moreover, to address the utilization problems mentioned above for centralized schedulers, we extended YARN by allowing tasks to be queued at each node, thus masking task allocation delays. In this case, the RM assigns tasks to node queues in a way that is similar to how it already assigns tasks to nodes. The resulting job completion times are depicted in the "YARN+Q" line of Figure 2.2. We see that naively offering FIFO queues at worker nodes in YARN can be worse than not having queues at all. As will be shown later, this is due to similar head-of-line issues, as well as the potentially poor early binding of tasks to machines.

[3] For example, when scheduling a task of job_1 with equal preference for machines $\{m_1, m_2\}$ and a task of job_2 that will run much faster at m_1, it is not possible to guarantee that job_2's task will always run at m_1.

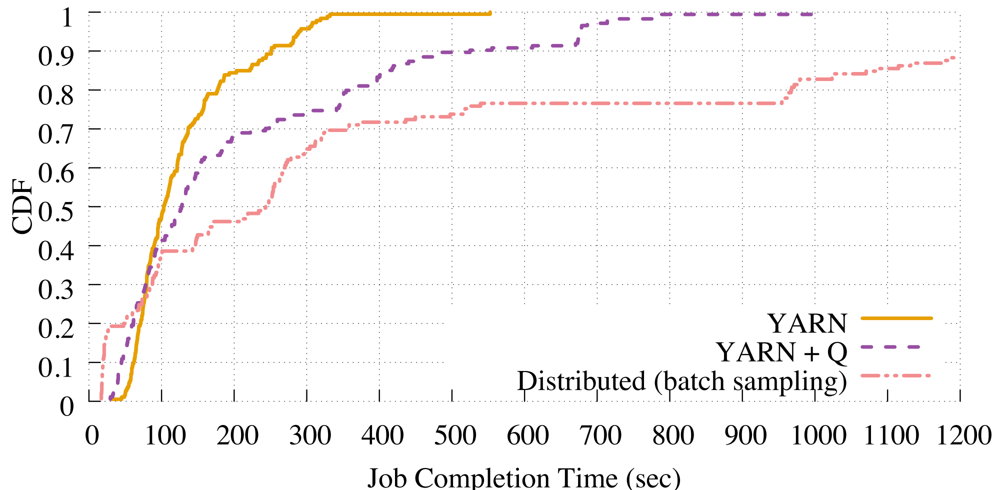

Figure 2.2: Job completion times for production Workload 2 using different scheduling types.

The remaining sections in this chapter describe the design of two cluster scheduler variations, *Yaq-c* and *Yaq-d*, upon which we implement and evaluate our application-aware queue management techniques. Yaq-c extends the centralized scheduler in YARN [59] by adding task queues at worker nodes. Yaq-d, on the other hand, is a distributed scheduler that extends the Mercury scheduler [35, 63]. After laying out the requirements for our scheduler (section 2.2), we first give an overview of our queuing techniques and compare Yaq's capabilities with those of existing scheduling frameworks (section 2.3). Then, we present the basic components of our system design (section 2.4), and detail the specifics of our design for both Yaq-c and Yaq-d (section 2.5 and section 2.6).

2.2 Requirements

Resource managers for large shared clusters need to meet various, often conflicting, requirements. Based on conversations with cluster operators and users, we distill the following set of requirements for our system.

Heterogeneous jobs: Due to workload consolidation, production clusters have to simultaneously support different types of jobs and services (e.g., production jobs, best-effort jobs). Hence,

	Scheduling type	Type of queuing		Queue management		
Scheduling framework	Centralized/Distributed	Global queue	Queues at nodes	Task placement	Queue sizing	Queue reordering
YARN [59]	✓/-	✓		*		
Borg [60]	✓/-	✓		*		
Sparrow [44]	-/✓		✓	✓		
Apollo [13]	-/✓		✓	✓		
Mercury [35]	✓/✓	✓	✓	✓		
Yaq-c	✓/-	✓	✓	✓	✓	✓
Yaq-d	-/✓		✓	✓	✓	✓

Table 2.2: Overview of queuing capabilities of existing scheduling frameworks compared to Yaq ($*$ indicates that the system performs placement of tasks to nodes but not to queues).

tasks have highly variable durations and resource needs (e.g., batch jobs, ML, MPI, etc.).

High cluster utilization: Since cluster operators seek to maximize return on investment, the scheduler should optimally use the cluster resources to achieve high cluster utilization. The expectation is that higher cluster utilization leads in turn to higher task and job throughput.

Fast (and predictable) job completion time: Cluster users desire that their jobs exit the system quickly, perhaps as close as possible to the jobs' ideal computational time. Furthermore, predictable completion times can substantially help with planning.

Sharing policies: Since the cluster is shared amongst multiple users, operators require support for sharing policies based on fairness and/or capacity constraints.

2.3 Task Queuing Overview

As we will describe in Chapter 3, the introduction of local queues in Yaq-c, and the application-aware management of the different queues in both Yaq-c and Yaq-d are our key contributions. It is thus useful to contrast our designs with existing systems.

In Table 2.2, we outline the type of queuing that existing systems enable (global queuing and/or local at the nodes), as well as the queue management capabilities they support compared to Yaq. Due to their inherent design, distributed and hybrid schedulers (such as Sparrow, Apollo, Mercury) support queuing at the nodes, but not global job queuing. On the other hand, to the best of our knowledge, no existing centralized system supports queuing at worker nodes. This is an interesting

point in the design space that we explore in Yaq-c. Further, although all systems with queues at worker nodes need to implement a task placement policy, none of them implement additional queue management techniques, such as task prioritization through queue reordering, and queue sizing. Hence, we explore such techniques in Yaq-c and Yaq-d.

2.4 Basic System Components

The general system architecture, depicted in Figure 2.3 (for Yaq-c) and Figure 2.4 (for Yaq-d), consists of the following main components:

Node Manager (NM) is a service running at each of the cluster's worker nodes, and is responsible for task execution at that node. Each NM comprises *running tasks* and *queued tasks* (as shown in Figures 2.3 and 2.4). The former is a list with the tasks that are currently being executed, thus occupying actual resources at the node. The latter is a queue with the tasks that are waiting on the resources held by the currently running tasks and are thus not occupying actual resources. A task is queued only if the NM cannot start its execution due to insufficient resources.

Resource Manager (RM) is the component that manages the cluster resources in centralized scheduling settings (thus appears only in Yaq-c). The NMs periodically inform the RM about their status through a heartbeat mechanism. Based on the available cluster resources and taking into account various scheduling constraints (e.g., data locality, resource interference, fairness/capacity) and a queue placement policy (to determine where tasks will be queued, if needed), it assigns resources to tasks for execution.

Usage Monitor (UM) is a centralized component to which the NMs periodically report their status. It is used in distributed scheduling frameworks as a form of loose coordination to perform more educated scheduling decisions. Although this component is not necessary [44], a form of a UM has been used in existing distributed schedulers [13, 35], and is also used in Yaq-d.

Job Manager (JM) is a per-job orchestrator (one JM gets instantiated for each submitted job).

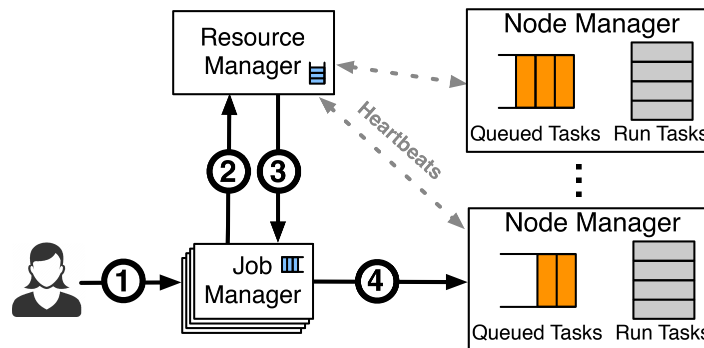

Figure 2.3: System architecture for centralized scheduling (Yaq-c).

In centralized settings, it negotiates with the RM framework for cluster resources. Once it receives resources, it dispatches tasks for execution to the associated nodes. In distributed settings, where there is no central RM, it also acts as a scheduler, immediately dispatching tasks to nodes.

2.5 Centralized Scheduling With Queues (Yaq-c)

Our system architecture for centralized scheduling is depicted in Figure 2.3. As shown in the figure, a job's lifecycle comprises the following steps. First, as soon as a client submits a new job to the cluster, the JM for this job gets initialized (step 1). The tasks of the job get added to the queue that is maintained locally in the JM. Then, the JM petitions the RM for cluster resources based on the resource needs of the job's tasks (step 2). The RM chooses where to place the tasks based on some policy (such as resource availability, status of queues at the NMs, data locality, etc.), and then notifies the JM (step 3). Subsequently, the JM dispatches the tasks for execution at the specified nodes (step 4). A task will start execution whenever it is allocated resources by the NM, and until that moment it is waiting at the NM's queue. The job's lifecycle terminates when all of its tasks complete execution.

Note that the RM performs job admission control, based on the available resources and other constraints (e.g., cluster sharing policies). Thus, when a job is submitted, it waits at a global queue in the RM (shown in blue in the figure), until it gets admitted for execution.

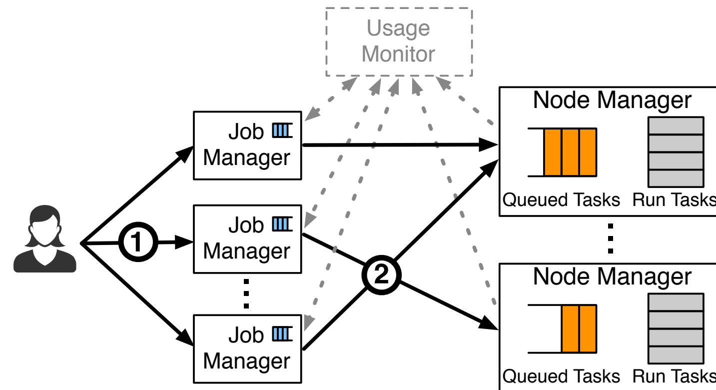

Figure 2.4: System architecture for distributed scheduling (Yaq-d).

2.6 Distributed Scheduling With Queues (Yaq-d)

Our system architecture for distributed scheduling is shown in Figure 2.4. When a client submits a new job, the corresponding JM gets instantiated (step 1 in the figure). The JM, who is now acting as the task scheduler for that job, uses a scheduling policy to select the node to which each of the job's task will be dispatched. The scheduling policy relies also on information that becomes available from the UM, such as the queue status of a node. The JM then places the tasks to the specified nodes for execution (step 2). Similar to the centralized case, if resources in a node are available, task execution starts immediately. Otherwise, the task waits in the queue until resources become available.

Our design also enables restricting the number of concurrently executing or queued tasks per JM. We defer the details to section 4.1.

Chapter 3

Worker Node and Global Queue Management

In the design outlined so far, queues at worker nodes are of particular importance since they determine when a task bound to a node starts execution. This is the case with either architecture, centralized or distributed. However, as explained in section 2.1, simply maintaining a queue of tasks waiting for execution at worker nodes does not directly translate to benefits in job completion time, especially in the presence of heterogeneous jobs.

Our primary focus in this work is on efficiently managing local node queues, we defer discussion of global queue management techniques to section 3.4. Our local queue management includes the following techniques: (1) determine the queue length (section 3.1); (2) decide the node to which each task will be placed for queuing (section 3.2); (3) prioritize the task execution by reordering the queue (section 3.3). We discuss cluster-wide queue management policies in section 3.4.

Note that placing tasks to queues is required whenever the actual cluster resources are not sufficient to accommodate the jobs that are submitted in the cluster. Thus, our techniques become essential under high cluster load. In cases of lower cluster load, when no worker-side queuing is needed, Yaq-c behaves like YARN and Yaq-d like Mercury.

To simplify our analysis, in this section we consider *slots* of resources consisting of memory and CPU, as done in YARN too. Whenever applicable, we discuss how our techniques can be extended

to support multi-dimensional resources.

Task duration estimates Part of our work relies on estimates of task durations, based on the observation that in our production clusters at Microsoft, more than 60% of the jobs are recurring. For such jobs, we assume an initial estimate of task durations based on previous executions. As we show in our experiments, Yaq performs well even with only rough estimates (such as the average duration of a map or reduce stage).[1] In the absence of such estimates, we assume a default task duration and have extended the JM to observe actual task durations at runtime and refine the initial estimate as the execution of the job proceeds.

3.1 Bounding Queue Lengths

Determining the length of queues at worker nodes is crucial: queues that are too short lead to lower cluster utilization, as resources may remain idle between allocations; queues that are too long may incur excessive queuing delays. We discuss two mechanisms for bounding queue lengths: length-based bounding (subsection 3.1.1) and delay-based bounding (subsection 3.1.3).

3.1.1 Length-Based Queue Bounding

In length-based queue bounding, all nodes have a predefined queue length b, and the RM can place up to b tasks at the queue of each node. We now focus on how to determine the value of b. We first consider the case when all tasks have the same duration, and then turn to the more general case.

Note that we base our analysis on the centralized design, where task placement is heartbeat-driven. We defer the analysis for the distributed case for future work, but expect the findings to be largely similar.

Fixed task duration We assume that all tasks have the same duration $1/\mu$ (where μ is the task processing rate), and calculate the minimum queue length that would guarantee a desired cluster utilization. Let r be the maximum number of tasks that can run concurrently at a node (based on

[1] Note that more sophisticated models for estimating task durations can be employed. We purposely opted for a simpler approach here, to assess our system's behavior even with inaccurate estimates.

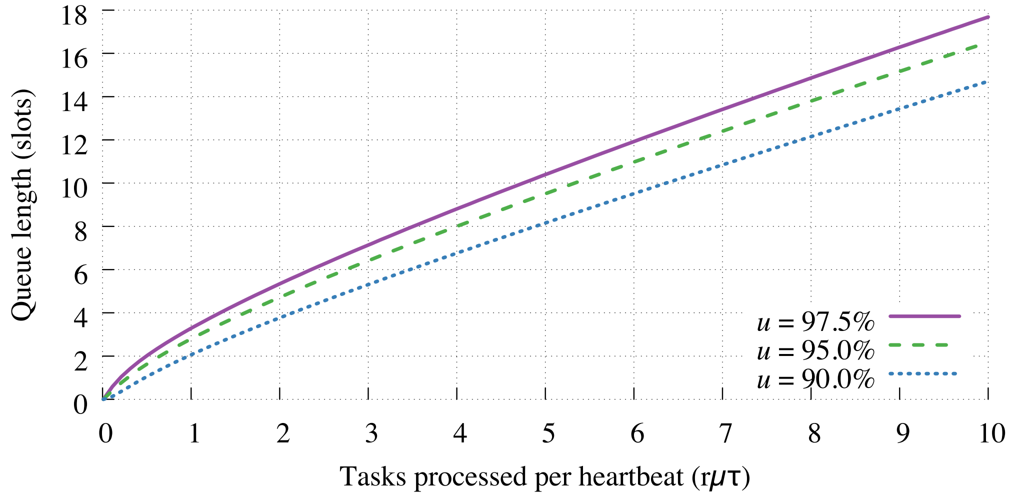

Figure 3.1: Queue length required to achieve cluster utilization $u=1-\delta$, given the number of tasks that can be processed by a node per heartbeat interval $(r\mu\tau)$.

its resources and the minimum resource demand of a task), and τ the heartbeat interval. Then the maximum task processing rate at the node is $r\mu$. Given r running tasks and b queued tasks, a node will remain fully utilized when: $r + b \geq r\mu\tau$ or $b \geq r(\mu\tau - 1)$.

Interestingly, the above reasoning is similar to the bandwidth-delay product for TCP flows, where the goal is to have enough packets in flight to keep the link fully utilized. In cluster scheduling, tasks can be seen as analogous to packets, node processing rate to the link capacity, and heartbeat interval to RTT.

Exponentially-distributed task duration We consider an arbitrary node that has r run slots and a queue of length b slots. We want to determine the value of parameter b such that node utilization is at least $1 - \delta$ for given parameter $\delta \in (0, 1]$. Here we provide the main results of our analysis; more details along with our proofs can be found in subsection 3.1.2. We note that node utilization is at least as large as the fraction of heartbeat intervals in which all run slots are always busy. It thus suffices to configure the queue length so that the latter quantity is at least of value $1 - \delta$.

We admit the following assumptions. Whenever the node completes processing a task, we assume that it starts processing one of the tasks from the queue taken uniformly at random, if there are any in the queue. We assume that task processing times are independent and identically distributed

according to an exponential distribution with mean $1/\mu$. This assumption enables us to characterize the node utilization by leveraging the memory-less property of the exponential distribution.

Proposition 1. *At least a $1-\delta$ fraction of heartbeat intervals will have all run slots always busy, if the queue length b is at least as large as the smallest integer that satisfies*

$$r\mu\tau\left(1+\frac{b+1}{r\mu\tau}\left(\log\left(\frac{b+1}{r\mu\tau}\right)-1\right)\right)\geq\log\left(\frac{1}{\delta}\right). \tag{3.1}$$

We now discuss the asserted sufficient condition. If the task processing times were deterministic assuming a common value $1/\mu$ and the length of the heartbeat interval is a multiple of $1/\mu$, then for 100% utilization it is necessary and sufficient to set the queue length such that $b+r=r\mu\tau$. This yields the queue length that is linear in $r\mu\tau$, for any fixed value of the run slots r. The sufficient condition in (3.1) requires a larger queue length than $r\mu\tau$ for small values of $r\mu\tau$. It can be shown that the sufficient condition (3.1) requires the queue length that is at least $r\mu\tau+\sqrt{\log(1/\delta)}\sqrt{r\mu\tau}$, for large $r\mu\tau$. For numerical examples, see Figure 3.1. Specifically, given a heartbeat interval $\tau=3$ sec, an average task duration $1/\mu$ of 10 sec, $r=10$ tasks allowed to be executed at a node at the same time, and a target utilization of 95%, a queue of $b=6$ slots is required. Likewise, for an average task duration of 30 sec, the queue size should be ≥ 3 slots. These values for b are also validated by our experiments (section 4.2) on the production Workload 2 of Figure 2.1.

3.1.2 Determining the Queue Length

We consider an arbitrary node that has r run slots and a queue of length b slots. We want to determine the value of the parameter b such that the utilization of the node is at least $1-\delta$ for given parameter $\delta\in(0,1]$.

We admit the following assumptions. Let τ be the length of a heart-beat interval. The node is fed with new tasks at the beginning of each heart-beat interval such that there are at most r tasks being processed by the node and at most b tasks being queued for processing at the node. Whenever the node completed processing a task it starts processing one of the tasks from the queue taken uniformly at random, if there is any in the queue. We assume that task processing times are

independent and identically distributed according to exponential distribution with mean $1/\mu$. This assumption enables us to characterize the node utilization by leveraging the memory-less property of the exponential distribution.

The node *utilization* is denoted with u and is defined as the average fraction of time the run slots of the node are busy processing tasks over an asymptotically large time interval. More formally, let $Q_i(t) = 1$ if at time t run slot i is busy, and $Q_i(t) = 0$, otherwise. Then, the node utilization is defined by

$$u = \lim_{T\to\infty} \frac{1}{T} \int_0^T \frac{\sum_{i=1}^r \mathbf{1}(Q_i(t) = 1)}{r} dt.$$

where $\mathbf{1}(A) = 1$ if condition A is true, and $\mathbf{1}(A) = 0$, otherwise.

Let $X_{n,\lambda}$ be a random variable with distribution that corresponds to the sum of n independent random variables with exponential distribution of mean $1/\lambda$. Note that the distribution of $X_{n,\lambda}$ is Erlang distribution with parameters n and λ, wich has the density function

$$f_{n,\lambda}(x) = \frac{\lambda^n x^{n-1} e^{-\lambda x}}{(n-1)!}, \text{ for } x \geq 0.$$

Proposition 2. *Under the given assumptions, the node utilization is given by*

$$\begin{aligned} u &= 1 - \left(\left(1 - \frac{1}{\mu\tau}\right) \Pr[X_{b,r\mu} \leq \tau] \right. \\ & \left. + \frac{1}{\mu\tau} \frac{e^{-\mu\tau}}{\left(1-\frac{1}{r}\right)^b} \Pr[X_{b,(r-1)\mu} \leq \tau] \right) \end{aligned}$$

Proof. We are interested in the node utilization with respect to the stationary distribution. Suppose that time 0 is the beginning of a heart-beat interval. We can use the Palm inversion formula (or ”cycle formula”) to note that the node utilization is equal to

$$u = \frac{\sum_{i=1}^r \mathbb{E}\left[\int_0^\tau \mathbf{1}(Q_i(t) = 1) dt\right]}{r\tau}.$$

It suffices to consider an arbitrary run slot i of the node and characterize the expected value of $\int_0^\tau \mathbf{1}(Q_i(t) = 1) dt$. By the memory-less property of the exponential distribution, there are $r + b$

tasks at time 0 whose (residual) processing times are independent and have exponential distribution with mean $1/\mu$. Whenever there are r tasks being processed by the node, the earliest time until completion of a task is equal in distribution to a minimum of r independent exponentially distributed random variables each with mean $1/\mu$; hence, it has exponential distribution with mean $1/(r\mu)$. It follows that the earliest time at which the queue is empty is equal in distribution to $X_{b,r\mu}$. From this time instance, each run slot completes processing the task assigned to it after an independent random duration that has exponential distribution with mean $1/\mu$. From this discussion, we conclude that

$$\mathbb{E}\left[\int_0^\tau \mathbf{1}(Q_i(t) = 1)dt\right] = \tau \Pr[X_{b,r\mu} > \tau]$$
$$+ \int_0^\tau \mathbb{E}[\min\{\sigma, \tau - x\}] d\Pr[X_{b,r\mu} \leq x]$$

where σ is a random variable with exponential distribution with mean $1/\mu$.

By a simple calculus, we have

$$\begin{aligned}
\mathbb{E}[\min\{\sigma, t\}] &= \int_0^\infty \Pr[\min\{\sigma, t\} > x]dx \\
&= \int_0^t \Pr[\sigma > x]dx \\
&= \int_0^t e^{-\mu x}dx \\
&= \frac{1}{\mu}\left(1 - e^{-\mu t}\right).
\end{aligned}$$

Hence, it follows that the utilization is given by

$$u = \Pr[X_{b,r\mu} > \tau]$$
$$+ \frac{1}{\mu\tau}\int_0^\tau \left(1 - e^{-\mu(\tau - x)}\right) d\Pr[X_{b,r\mu} \leq x]$$

which by some elementary calculus can be written as asserted in the proposition. □

Notice that, in particular, for a node with zero queue slots,

$$u = \frac{1 - e^{-\mu\tau}}{\mu\tau}.$$

A simple lower bound on the node utilization can be derived as follows. Let A_k denote the event that in the k-th heart-beat interval none of the run slots is every idle. Notice that

$$u \geq \Pr[A_k].$$

The event A_k is equivalent to the event that the time elapsed from the k-th heart beat until the completion of the $(b+1)$-st task, among the tasks present just after the k-th heart beat, is larger than the length of the heart-beat interval τ. Notice that the distribution of this time duration is equal Erlang distribution with parameters $b+1$ and $r\mu$. Hence, we have

$$\Pr[A_k] = \Pr[X_{b+1,r\mu} > \tau].$$

It follows that a sufficient condition for the node utilization to be at least $1-\delta$ is the following condition

$$\Pr[X_{b+1,r\mu} \leq \tau] \leq \delta. \tag{3.2}$$

Proposition 3. *A sufficient condition for the probability that in a heart-beat interval none of the run slots is ever idle is at least $1-\delta$ is that the queue length b is the smallest integer such that it holds*

$$r\mu\tau\left(1 + \frac{b+1}{r\mu\tau}\left(\log\left(\frac{b+1}{r\mu\tau}\right) - 1\right)\right) \geq \log\left(\frac{1}{\delta}\right). \tag{3.3}$$

Before giving a proof of the proposition, we discuss the asserted sufficient condition. If the task processing times were deterministic assuming a common value $1/\mu$ and the length of the heart-beat interval is a multiple of $1/\mu$, then for 100% utilization it is necessary and sufficient to set the queue length such that $b + r = r\mu\tau$. This yields the queue length that is linear in $r\mu\tau$, for any fixed value of the run slots r. The sufficient condition in (3.3) requires a larger queue length than $r\mu\tau$ for small

values of $r\mu\tau$. It can be shown that the sufficient condition (3.3) requires the queue length that is at least $r\mu\tau + \sqrt{\log(1/\delta)}\sqrt{r\mu\tau}$, for large $r\mu\tau$.

For numerical examples, see Figure 3.1. Specifically, given a heartbeat interval $\tau = 3$ sec, an average task duration $1/\mu$ of 10 sec, $r = 10$ tasks allowed to be executed at a node at the same time, and a target utilization of 95%, a queue of $b = 6$ slots is required. Likewise, for an average task duration of 30 sec, the queue size should be ≥ 3 slots. These values for b are also validated by our experiments (section 4.2) on the production Workload 2 of Figure 2.1.

Proof. The proof follows by (3.2) and the Chernoff's inequality, which we describe as follows.

We first establish the following claim:

$$\Pr[X_{n,\lambda} \leq x] \leq e^{-\lambda x\left(1+\frac{n}{\lambda x}\left(\log\left(\frac{n}{\lambda x}\right)-1\right)\right)}, \text{ for } x \geq 0. \tag{3.4}$$

Let $\sigma_1, \sigma_1, \ldots, \sigma_n$ be a sequence of independent exponentially distributed random variables each of mean $1/\lambda$. Using Chernoff's inequality, for every $\theta > 0$, we have

$$\begin{aligned}
\Pr[X_{n,\lambda} \leq x] &\leq e^{\theta x}\mathbb{E}[e^{-\theta\sum_{i=1}^{n}\sigma_i}] \\
&= e^{\theta x}\prod_{i=1}^{n}\mathbb{E}[e^{-\theta\sigma_i}] \\
&= e^{\theta x}\mathbb{E}[e^{-\theta\sigma_1}]^n \\
&= e^{\theta x}\left(\frac{\lambda}{\lambda+\theta}\right)^n.
\end{aligned}$$

The minimizer of the last expression is for the value of parameter θ such that

$$\lambda x + \theta x = n.$$

Hence, we obtain the inequality asserted in (3.4).

Using (3.4), have

$$\Pr[X_{b+1,r\mu} \leq \tau] \leq e^{-r\mu\tau\left(1+\frac{b+1}{r\mu\tau}\left(\log\left(\frac{b+1}{r\mu\tau}\right)-1\right)\right)}.$$

By requiring that the right-hand side in the last inequality is smaller than or equal to δ, we obtain the inequality asserted in the proposition.

For every integer value b such that condition (3.3) holds, we have that $\Pr[X_{b+1,r\mu} \leq \tau] \leq \delta$, which implies the node utilization of at least $1 - \delta$. Since the left-hand side of the inequality in (3.3) is increasing in b, it suffices to chose the queue length that is the smallest integer b such that condition (3.3) holds. $\square$

3.1.3 Delay-Based Queue Bounding

Maintaining queues of the same fixed length across all nodes does not work well with heterogeneous tasks. When short tasks happen to be present in a node, this may lead to under-utilization of its resources, whereas when tasks are longer, significant delays may incur. Hence, when task durations are available, we use a *delay-based* strategy. This strategy relies on the estimated queue wait time that gets reported by each node at regular intervals, as we explain in section 3.2 (Algorithm 2). In particular, we specify the maximum time WT_{max} that a task is allowed to wait in a queue. When we are about to place a task t at the queue of node n (see section 3.2), we first check the last estimated queue wait time WT_n reported by n. Only if W$T_n < WT_{max}$ is t queued at that node. Upon queuing, the RM uses a simple formula to update WT_n taking into account t's task duration estimate, until a fresh value for WT_n is received from n. Using this method, the number of tasks that get queued to each node gets dynamically adapted, based on the current load of the node and the tasks that are currently running and queued.

Note that this technique can be directly applied in both our centralized and distributed designs.

3.2 Placement of Tasks to Nodes

Given a job consisting of a set of tasks, the scheduler has to determine the nodes to which those tasks will be placed. We now present the algorithm that Yaq uses for task placement. We also introduce the algorithm that we use to estimate the time a task has to wait when placed in a node's queue before starting its execution. This algorithm is crucial for high quality task placement decisions.

Algorithm 1: Placement of task to node

Input : t: task to be placed; Rf_{min}: min free cluster resources percentage before starting to queue tasks

Output: node where t will be placed

```
    // Avoid queuing when available cluster resources
1   if freeResources / totalResources > Rf_min then
2       return placeTaskNoQueuing(t)
3   else return node n with highest queuingScore(n, t)
    ----------------------------------------------------------------
    // How suitable is node n for placing task t to its queue
4   Function queuingScore(n, t)
        // affScore ∈ (0, 1] based on data locality (or resource interference) when placing t on
           n (higher is better)
5       affScore ← affinityScore(n, t)
        // Compute load of node based on queue length or queue wait time (using Algorithm 2)
6       nload ← nodeLoad(n)
7       return affScore × 1/nload
```

As explained in chapter 2, we assume that there is a central component to which each node periodically publishes information about its resource and queue status. This component is the RM in Yaq-c (see Figure 2.3), and the UM in Yaq-d (see Figure 2.4).

Our task placement algorithm is outlined in Algorithm 1. It takes as input a task t and outputs the node n where t should be placed. Yaq preferentially places tasks at nodes that have available resources since such tasks will incur no queuing delays. Thus, we first check if there are such resources (line 1), and if so, we place t to a node with available local resources, taking other parameters such as data locality also into account (line 2). If the cluster is almost fully loaded (as defined by the Rf_{min} parameter given as input), we choose which node's queue to place t (line 3). We use the function `queuingScore(n,t)` to quantify how suitable a node is for executing t. The score of a node comprises two components: node affinity for t and node load. In our current implementation, node affinity takes into account data locality, but this can be extended to also consider resource interference, providing better resource isolation when executing t. The load of the node can be calculated based on one of the following strategies depending on the richness of the information published by each node:

Based on queue length: The simplest information that each node publishes is the size of its queue.

Algorithm 2: Estimate queue wait time at node

Input : $runTasks$: running tasks' remaining durations; $queuedTasks$: queued tasks' durations; $freeResources$: free node resources; $freeResources_{min}$: min free node resources before considering a node full

Output: Estimated queue wait time for the next task that will be dispatched to the node

```
1 if freeResources ≥ freeResources_min then
2 |   return 0
3 waitTime ← 0
4 for qTask in queuedTasks do
5 |   minTask ← remove_min(runTasks)
6 |   waitTime ← waitTime + minTask
7 |   runTasks ← [t − minTask for t in runTasks]
8 |   runTasks.add(qTask)
9 return waitTime + remove_min(runTasks)
```

This strategy gives higher score to nodes with smaller queue lengths. Note that this can lead to suboptimal placement decisions in case of heterogeneous tasks: a node with two queued tasks of 500 secs each will be chosen over a node with five tasks of 2 secs each.

Based on queue wait time: This strategy assumes that each node publishes information about the estimated time a task will have to wait at a node before starting its execution, as described below. The lower this estimated wait time is, the higher the score of the node. This strategy improves upon the previous one when considering heterogeneous tasks, as we also show experimentally in subsection 4.7.2.

Note that Algorithm 1 suggests that we calculate the score of all nodes for placing each task. Clearly this can lead to scalability issues, thus in practice we apply various optimizations (e.g., compute the score of each node only at regular intervals).

Estimating queue wait time at worker nodes Algorithm 2 outlines how each worker node independently estimates the expected queuing delay that a new task will incur if it is placed in its queue. Queue wait time estimates are then periodically sent to the RM (in Yaq-c) or UM (in Yaq-d) to help

with task placement. Effectively, the algorithm simulates CPU scheduling. It takes as input the remaining durations of the currently running tasks, and the durations of the queued tasks.[2] If there are available resources, the new task will not have to wait (line 2). Otherwise, we go over the queued tasks and accumulate the time that each of them has to wait before its execution starts (lines 4-8). The first task in the queue will have to wait for the running task with the smallest remaining duration to finish. Then that task gets removed from the running task list (line 5), and its task duration gets added to the accumulated queue wait time (line 6). All remaining running task durations get updated (line 7), the first task in the queue gets added to the list of running tasks (line 8), and the same process repeats for all queued tasks.

Observe that in our estimation, we make the assumption that a queued task can take the slot of any previously running task. One could extend our algorithm to take into account the exact resources required by each task, similar to the queue-wait time matrix of Apollo [13].

Observe that from the moment a task gets placed to a node's queue until the moment its execution starts, better placement choices may become available. This may be due to incorrect information during initial task placement (e.g., wrong queue load estimates) or changing cluster conditions (e.g., resource contention, node failures). Various *corrective actions* can be taken to mitigate this problem, such as dynamic queue rebalancing [35], duplicate execution [4, 13, 17] or work stealing [18]. Since duplicate execution hurts effective cluster utilization, and work stealing makes it hard to account for locality and security constraints in a shared cluster, in Yaq we use queue rebalancing. However, any other technique could be used instead.

3.3 Prioritizing Task Execution

The queue management techniques presented so far are crucial for improving task completion time: they reduce queuing delay (section 3.1) and properly place tasks to queues (section 3.2). However, as we also show experimentally in section 4.2, they do not improve job completion time on their own most of the time. This is because they execute queued tasks in a FIFO order, without taking into

---

[2]These are *estimations* of task durations, as explained in the beginning of chapter 3.

Algorithm 3: Compare task priorities

```
  Input  : tasks t1, t2; comparison strategy taskCmp; hard starvation threshold ST; relative
           starvation threshold ST_r
  Output: > 0 if t1 has higher priority, < 0 if t2 has higher priority, else 0
1  if isStarved(t1) xor isStarved(t2) then
2      if isStarved (t1) then return +1
3      else return −1
4  if !isStarved(t1) and !isStarved(t2) then
5      cmp ← taskCmp(t1, t2)
6      if cmp ≠ 0 then return cmp
7  if isStarved(t1) and isStarved(t2) then
8      cmp ← t2.jobArrivalTime − t1.jobArrivalTime
9      if cmp ≠ 0 then return cmp
10 return t1.queueTime − t2.queueTime
   ---------------------------------------------------------------
11 Function isStarved (ti)
12     return ti.queueTime > min(ST, ti.durationEst × ST_r)
```

account the characteristics of the tasks and of the jobs they belong to. To this end, we introduce a task prioritization algorithm that reorders queued tasks and can significantly improve job completion times (see section 4.2).

Our prioritization algorithm is generic in that any *queue reordering strategy* can be easily plugged in. Moreover, it is *starvation-aware*, guaranteeing that no task will be starved due to the existence of other higher priority tasks. We implemented various reordering strategies, which we present below. Among them, a significant family of strategies are the *job-aware* ones, which consider *all of the tasks in a job* during reordering. In particular, we emphasize on Shortest Remaining Job First that gave us the best results in our experiments.

Our task prioritization algorithm is outlined in Algorithm 3. It takes as input two tasks, one of the reordering strategies `taskCmp` (among the following, which we detail below: SRJF, LRTF, STF, EJF), as well as a hard and a relative starvation threshold. Tasks are marked as starved, as explained below, using the function `isStarved` (lines 11-12). Starved tasks have higher priority than non-starved ones (lines 1-3). If none of the tasks are starved, we compare them with `taskCmp` (lines 4-6). If both are starved, we give higher priority to the task of the earlier submitted job (lines 7-9). We finally break ties by comparing the time each task has waited in the queue.

Queue reordering strategies: We have implemented and experimented with the following reordering strategies:

Shortest Remaining Job First (SRJF) gives highest priority to the tasks whose jobs have the least remaining work. The remaining work for a job j is a way to quantify how close j is to completion. It is computed using the formula $\Sigma_{t_i \in RT(j)} t_i \cdot td(t_i)$, where $RT(j)$ are the non-completed tasks of j and $td(t_i)$ is the (remaining) task duration of task t_i, based on our task duration estimates. The remaining work gets propagated from the RM (in Yaq-c) or the UM (in Yaq-d) to the nodes through the existing heartbeats.

Least Remaining Tasks First (LRTF) is similar to SRJF, but relies on number of remaining tasks to estimate the remaining work. Although this estimate is not as accurate as the one used by SRJF, it is simpler in that it does not require task duration estimates. The remaining tasks number gets propagated from the JM to the nodes through the existing heartbeats.

Shortest Task First (STF) orders tasks based on increasing expected duration. This strategy is the only one in this list that is not *job-aware*, given that it uses only task information and is agnostic of the status of the job the tasks belong to. However, it can become interesting when coupled with our starvation-aware techniques.

Earliest Job First (EJF) orders tasks based on the arrival time of the job that the tasks belong to. This is essentially FIFO ordering, and is the default strategy in most schedulers. No additional knowledge is required from the jobs. Although EJF performs no reordering, as described above, we use it to discriminate between starved tasks.

Commonalities between our reordering strategies and existing OS and network scheduling strategies are discussed in section 4.8. Observe that our current strategies are oblivious to the *job structure* (e.g., whether a task belongs to the map or reduce phase of a M/R job, the structure of a DAG job, etc.). As part of ongoing work, we are evaluating novel strategies that account for job structure which can be used to further prioritize task execution. Moreover, we are currently investigating how, in the presence of multi-dimensional resources, we can momentarily violate a reordering strategy in

order to provide better resource packing and thus achieve higher resource utilization.

Starvation-aware queue reordering All of the above strategies except EJF can lead to starvation or to excessive delays for some tasks. For example, long tasks can suffer with STF if short ones keep arriving. Similarly, tasks of *large* jobs can suffer with LRTF and SRJF. To circumvent this problem, during reordering we check whether a task has waited too long in the queue. If so, we give it higher priority. In particular, we specify both a hard (ST) and a relative (ST_r) threshold. A task is marked as starved if it has waited longer than ST_r times its duration or if it has waited longer than ST secs. ST_r allows short tasks to starve faster than long ones (e.g., a 2-sec task should be marked as starved sooner than a 500-sec task, but not more than ST secs).

3.4 Global Queue Management Policies

Our queue management techniques presented so far focused on task execution at specific nodes. We now discuss how Yaq can be coupled with cluster-wide policies. In particular, we first focus on techniques for global job reordering in the case of a centralized design (subsection 3.4.1), and then on imposing sharing policies, such as fairness and capacity (subsection 3.4.2).

3.4.1 Global Job Reordering

As discussed in section 2.5, Yaq-c maintains a queue of jobs at the RM.[3] Along with task reordering at each node, we can also devise *job* reordering strategies to be used at the RM. Similar to the task reordering strategies presented in section 3.3, we can apply SRJF, LRTF and EJF at the job level. More specifically, SRJF will give higher priority to jobs with the smallest remaining work, whereas LRTF will prioritize jobs with the least remaining number of tasks. EJF uses a FIFO queue. The analogous to STF would be SJF (Shortest Job First), assuming we have information about the job durations. Moreover, our starvation-aware techniques can be applied here as well, to avoid jobs from waiting too long in the RM queue. More advanced multi-resource packing techniques (such

[3]Notice that there can be no global job reordering in our distributed Yaq-d design, as there is no global queue in the system.

as Tetris [26]) can also be employed.

Although prioritizing jobs at the RM can be beneficial, what is more interesting in Yaq is how this global job reordering interplays with the local task reordering strategies, as they might have conflicting goals. For instance, when SRJF/LRTF are used both globally and locally, they are expected to further improve job completion times. However, this is probably not the case when SRJF is enabled globally and EJF locally: the former will favor jobs that are close to completion, whereas the latter will locally favor tasks with smaller duration. Our initial results show that there are indeed combinations that can further improve job completion times. We are currently working on formalizing such beneficial combinations, also taking into account workload characteristics.

3.4.2 Sharing Policies

Scheduling frameworks facilitate sharing of cluster resources among different users by imposing sharing policies. For instance, YARN [59] can impose both fairness (each user gets a fair share of the cluster) [7] and capacity (each user gets a percentage of the cluster) [6] constraints. Sparrow also shows how to impose weighted fair sharing in a distributed setting [44].

All these existing techniques can be applied in Yaq-c and Yaq-d in order to impose sharing constraints *over both running and queued tasks*. However, the scheduling framework has to impose constraints over the actual cluster resources (this is what the user actually observes). When task prioritization is disabled, the sharing constraints over the actual resources will be met, as each task will be executed in the order it was submitted by the scheduler. The problem arises in case of queue reordering: the scheduler has imposed constraints assuming a specific execution order, but this order might change, giving resources to the wrong users, thus exceeding their cluster share against others.

To circumvent this problem, we exploit the starvation threshold ST of our prioritization algorithm (see section 3.3). In particular, given that each task is marked as starved after ST seconds, actual resources will be given to it and sharing constraints will be met after that period of time.[4] As we experimentally show in section 4.6, Yaq-c is indeed able to successfully meet strict capacity

[4]As long as task preemption is enabled, otherwise a starved task has to wait for one of the running tasks to finish its execution.

constraints with only slight momentary violations.

Going further, we make the observation that the above technique is pessimistic in that it does not take advantage of user information about the queued tasks. If two tasks belong to the same user, they are not actually causing violation of sharing constraints between them. This can be solved by pushing auxiliary information about the users to worker nodes. Moreover, it is interesting to investigate whether task prioritization strategies can momentarily allow violations of sharing constraints in order to achieve better job completion times (using some form of deficit counters [51]).

Chapter 4

Evaluation of Yaq

4.1 Implementation

Yaq-c We implemented Yaq-c by extending Apache Hadoop's YARN [28] as follows. First, we extended YARN's NM to allow local queuing of tasks, and implemented our queue management techniques for bounding queue lengths (section 3.1) and prioritizing task execution (section 3.3). Second, we extended YARN's scheduler to enable placement of tasks to queues (section 3.2), support job prioritization (subsection 3.4.1), and respect cluster sharing constraints in the presence of task queuing (subsection 3.4.2). Finally, in the current implementation, we modified Hadoop's capacity scheduler [6], but our changes can be applied to any Hadoop-compatible scheduler (e.g., DRF [24], fair scheduler [7]).

Yaq-d We implemented Yaq-d by extending the distributed part of Mercury [35, 63] that already supports queuing at worker nodes. In particular, we implemented our techniques for task placement to queues and task prioritization on top of Mercury. In our current implementation, we do not bound the queue lengths, although that could be possible by allowing tasks to be queued at the JMs, in case no queue slots are available in a node. However, as our experimental results show, we already get significant gains over Mercury even without bounding queue lengths.

We have already made available the addition of task queues in YARN's NMs at Apache JIRA

YARN-2883 [64]. We also plan to open-source our queue management techniques both on YARN and Mercury.

4.2 Experimental Evaluation

The main results of our evaluation are the following:

- Yaq-c improves median job completion time (JCT) by 1.7x when compared to YARN over a production workload.
- Yaq-d, when evaluated over the same workload, achieves 9.3x better median JCT when compared to a scheduler that mimics Sparrow's batch sampling, and 3.9x better median JCT when compared to the distributed version of Mercury [35].
- Although task prioritization appears to provide the most pronounced benefits, the combination of all our techniques is the configuration that gives the best results.

Note that our purpose in this work is not to compare Yaq-c with Yaq-d. Instead, we want to study the performance improvement that Yaq-c and Yaq-d bring when compared to existing designs of the same type (centralized and distributed, respectively). Since they follow different architectures, each of them targets different scenarios: high level placement decisions and strict cluster sharing policies for Yaq-c; fast allocation latency and scalability for Yaq-d. Applying our techniques to hybrid schedulers [18, 35] would be an interesting direction for future work.

We now present results from our experimental evaluation. We first assess the performance of both Yaq-c (section 4.4) and Yaq-d (section 4.5) over a Hive production workload used at Microsoft, comparing our systems against existing centralized and distributed scheduling schemes. Then we show that Yaq-c can successfully impose sharing invariants (section 4.6). Lastly, we show a set of micro-experiments that highlight specific components of our design, such as queue-bounding, task placement, and task prioritization (section 4.7).

4.3 Experimental Setup

Cluster setup We deployed Yaq-c and Yaq-d on a cluster of 80 machines and used it for our evaluation. Each machine has a dual quad-core Intel Xeon E5-2660 processor with hyper-threading enabled (i.e., 32 virtual cores), 128 GB of RAM, 10 x 3 TB data drives configured as a JBOD. Inter-machine connectivity is 10 Gbps.

Our Yaq-c implementation is based on YARN 2.7.1. We use the same YARN version to compare against "stock" YARN. The Mercury implementation that we used was based on YARN 2.4.2, and the same holds for Yaq-d, since it was built by extending Mercury, as we explain in section 4.1. We also use Tez 0.4.1 [48] to execute all workloads, and Hive 0.13 for the Hive workload that is described below. In all our experiments, we use a heartbeat interval of 3 sec, which is also the typical value used in the YARN clusters at Yahoo! [59].

Workloads To evaluate Yaq-c and Yaq-d against other approaches, we use the Hive-MS workload, which is a Hive [57] workload used by an internal customer at Microsoft to perform data analysis. This is Workload 2 depicted in Figure 2.1. It consists of 185 queries, each having one map and one reduce phase. The underlying data consists of five relations with a total size of 2.49 PB. Each job has an average of 57.9 mappers and 1.5 reducers. Tasks among all jobs have an average duration of 22.9 sec with a standard deviation of 27.8 sec, when run on stock YARN.

We also use synthetic GridMix [29] workloads, each consisting of 100 tasks/job executed for 30 min, where: (1) X sec is a homogeneous workload where all tasks in a job have the same task duration (e.g., 5 sec), (2) Mixed-5-50 is a heterogeneous workload comprising of 80% jobs with 5-second tasks and 20% jobs with 50-second tasks, and (3) GridMix-MS is another heterogeneous workload, in which task sizes follow an exponential distribution with a mean of 49 sec. GridMix-MS is based on Microsoft's production Workload 1, depicted in Figure 2.1, after scaling down the longer task durations to adapt them to the duration of our runs and the size of our cluster.

Moreover, in our experiments, the scheduler gets as input the estimated average task duration of the stage (map or reduce) each task belongs to, as observed by previous executions of the same

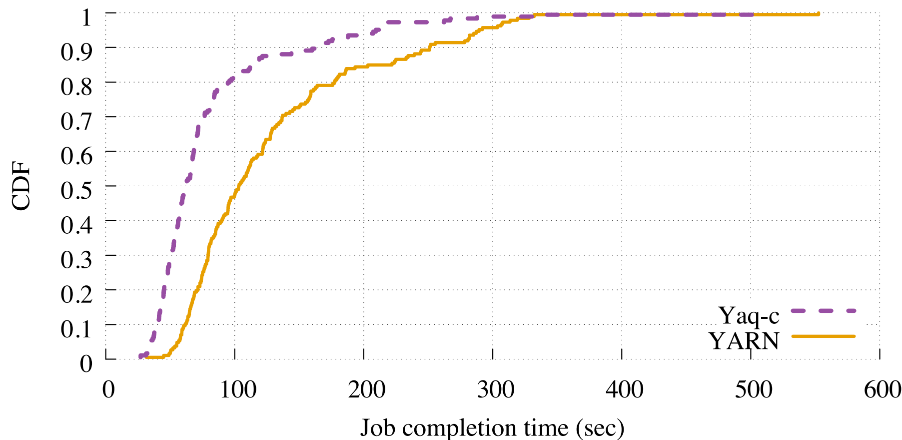

Figure 4.1: Job completion times for Yaq-c on Hive-MS workload.

job. Note that we deliberately provide such simple estimates, in order to assess Yaq under imprecise task durations. These estimates are then used during placement of tasks to nodes and for some of our task prioritization algorithms (see also discussion in the beginning chapter 3).

Metrics We base our analysis on the following metrics: *job completion time*, which is the time from the moment a job started its execution until the moment all tasks of the job finished execution; *slot utilization*, which is the number of slots[1] occupied at each moment across all machines, divided by the total number of slots in the cluster; *task queuing delay*, which is the time from the moment a task gets placed in a node's queue until its execution starts; *average job throughput*, which is the number of jobs in a workload, divided by the total time needed to execute all jobs, and is used to calculate effective cluster throughput.

4.4 Evaluating Yaq-c

To evaluate Yaq-c, we compare it against stock YARN. For Yaq-c we use a queue size of four slots (section 3.1), the queue wait time-based placement policy (section 3.2) and the SRJF prioritization policy (section 3.3), as those gave us the best results. The queue size we used coincides with the one suggested by our analysis using Equation 3.1. Figure 4.1 shows that Yaq-c achieves better

[1] We use 4 GB and 1 CPU per slot.

	Task queuing delay (sec)			**Job throughput**
	Mean	Stdev	Median	(jobs/min)
Yaq-c	8.5	21.4	1.1	13.9
Yaq-c (unbounded)	65.5	85.1	30.4	5.6
Yaq-c (no reorder)	53.2	78.2	25.4	7.6
YARN	-	-	-	8.8

Table 4.1: Average task queuing delay and job throughput for Yaq-c on Hive-MS workload.

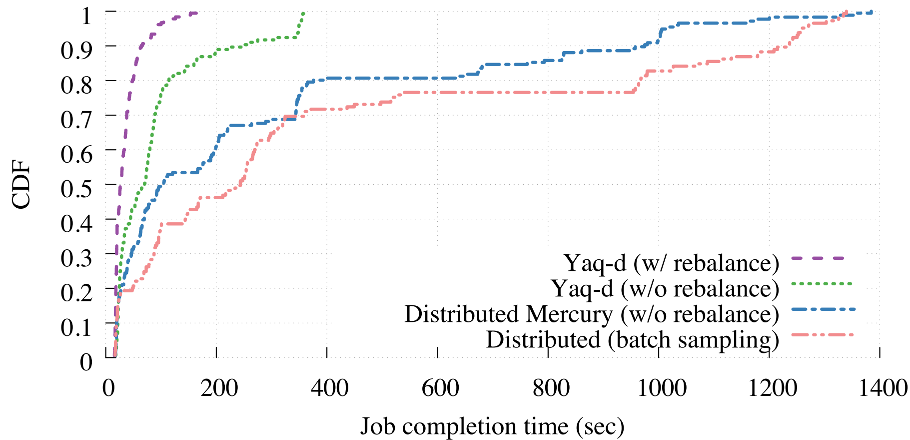

Figure 4.2: Job completion times for Yaq-d on Hive-MS workload.

job completion times across all percentiles with a 1.7x improvement for median job completion time. As shown in Table 4.1, Yaq-c also improves job throughput by 1.6x over YARN. These gains are due to the higher cluster utilization Yaq-c achieves by having worker-side queues (more details on utilization are given in subsection 4.7.1). Moreover, to show the benefit of our queue management techniques, in Table 4.1 we provide performance numbers for Yaq-c if we disable queue length bounding or task prioritization. In the absence of our techniques, we observe excessive task queuing delays that negatively impacts job throughput, also resulting in worse performance than stock YARN. On the contrary, Yaq-c achieves a median task queuing delay of only 1.1 sec.

4.5 Evaluating Yaq-d

We evaluate Yaq-d against two other distributed scheduler variants: *distributed Mercury*, which uses the distributed part of Mercury [35], and *distributed batch sampling*, for which we modified Mercury to perform task placement using batch sampling, as a way to simulate the task placement

	Task queuing delay (sec)			Job throughput
	Mean	Stdev	Median	(jobs/min)
Yaq-d (w/ rebalance)	17.9	54.2	0.35	16.6
Yaq-d (w/o rebalance)	34.2	67.0	5.6	10.1
Distributed Mercury	49.7	73.7	12.9	5.8
Distributed (batch sampl.)	81.4	101.4	26.2	5.3

Table 4.2: Average task queuing delay and job throughput for Yaq-d on Hive-MS workload.

done by Sparrow [44]. We use two different Yaq-d configurations with and without dynamic queue rebalancing (see end of section 3.2). Moreover, we use the queue wait time-based placement policy (section 3.2) and the SRJF prioritization policy with a 10-sec hard starvation threshold (section 3.3), which performed best in practice.

Our results for the Hive-MS workload are depicted in Figure 4.2 and Table 4.2. Yaq-d (with rebalance) improves job completion time (JCT) across all percentiles when compared to both Mercury and batch sampling. In particular, it achieves better median JCT by 3.9x over Mercury and by 9.3x over batch sampling. These improvements are due to the efficient management of the local queues, as we significantly reduce the task queuing delays and thus the head-of-line blocking.

Observe that in our Yaq-d implementation we do not use late binding of tasks to nodes, as it conflicts with some of YARN's design choices. As shown in Figure 8 of the Sparrow paper [44], late binding on top of batch sampling further improves average job completion time by 14% and the 95th percentile by $\sim$30%. Therefore, even if we implemented late binding, most probably Yaq-d would still significantly outperform Sparrow.

As can be seen from Table 4.2, Yaq-d also achieves a higher job throughput by 2.9x over Mercury and by 3.1x over batch sampling. When configuring Mercury and batch sampling, we had to tune the number of jobs that are allowed to be executed concurrently: allowing too many concurrent jobs improves job throughput but hurts JCT (due to having tasks belonging to many different jobs being queued at the nodes without properly sizing or reordering the queues); allowing fewer jobs improves JCT but leads to lower utilization and hurts job throughput. We could improve job throughput for Mercury and batch sampling in our runs by allowing more concurrent jobs, but that would lead to even worse JCT. On the contrary, Yaq-d improves both JCT and job throughput at the same time.

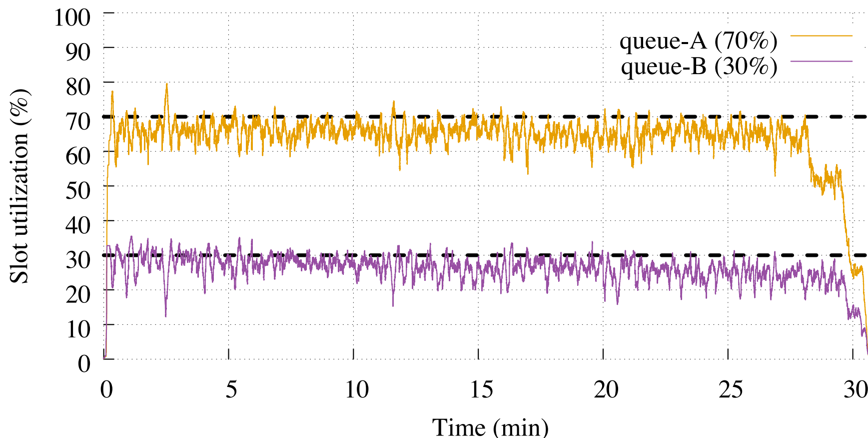

Figure 4.3: Imposing capacity constraints in Yaq-c.

4.6 Imposing Sharing Constraints

As discussed in subsection 3.4.2, task prioritization can potentially lead to violation of cluster-wide sharing policies. To this end, we use Yaq-c, whose implementation relies on Hadoop's capacity scheduler [6] (as explained in section 4.1) that is capable of imposing capacity quotas on each user of the cluster. To investigate whether Yaq-c continues to respect such cluster-wide sharing policies despite task prioritization, we configure the capacity scheduler with two queues, A and B, where the cluster capacity is split 70% and 30% respectively. We run a GridMix workload that submits jobs to both queues with equal probability. Figure 4.3 shows cluster-wide slot utilization for each of these two queues measured from the perspective of all worker nodes. As the figure shows, Yaq-c respects each queue's capacity with only some momentary slight violations.

4.7 Micro-experiments

We evaluate specific aspects of our queue management techniques using a set of micro-experiments. In these runs we use our synthetic GridMix workloads, which make it easier to experiment with different task duration distributions, whenever needed. We study the effects of bounding queuing

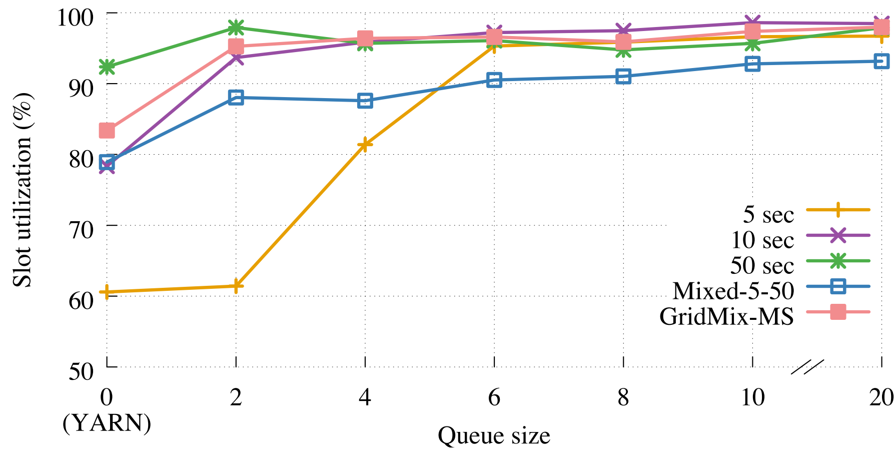

Figure 4.4: Average cluster slot utilization with different workloads and queue lengths.

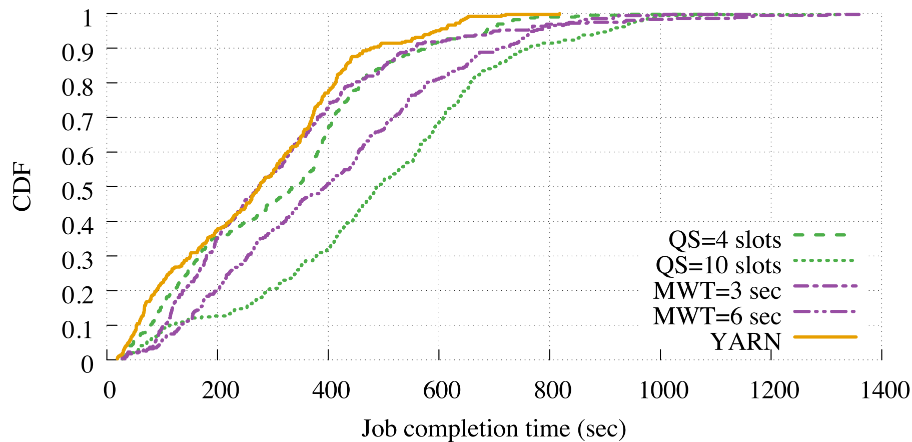

Figure 4.5: Job completion time for GridMix-MS workload with different queue bounding techniques and no task prioritization.

lengths (subsection 4.7.1), task placement choices (subsection 4.7.2), and task prioritization strategies (subsection 4.7.3). We also evaluate our techniques over a heavy-tailed distribution (subsection 4.7.4). Here we use Yaq-c, but we also observed similar trends with Yaq-d for task placement and prioritization.

4.7.1 Bounding Queuing Delays

We first study the impact of queue length in cluster utilization and job completion times (JCT). To this end, we purposely disable task prioritization in these experiments.

Figure 4.4 shows how slot utilization for Yaq-c varies for different workloads when introducing queuing at worker nodes. By masking feedback delays between the RM and NM, Yaq-c is able to prevent slots from becoming idle. The gains are particularly pronounced when task durations are short: for 5-sec tasks, average utilization is 60% with YARN but goes up to 96% with Yaq-c. The graphs also show that utilization improves with longer queue sizes, as expected. Furthermore, once the nodes are saturated increasing the queue sizes even further does not improve utilization. For instance, the 5 sec workload needs a queue size of six slots to achieve full utilization, while for the 50 sec workload a queue size of two slots is sufficient.

Figure 4.5 compares job completion time (JCT) of the GridMix-MS workload with YARN and both length-based (QS=x denotes that x tasks are allowed to be queued) and delay-based bounding (MWT=x denotes that queuing delay should not exceed x sec). For fixed queue lengths, we see that JCT increases with queue length. This is to be expected since increased queue lengths introduce higher queuing delays, without further improving utilization (as shown in Figure 4.4). Furthermore, the tail of the distribution also increases substantially when queue lengths increase, by upwards of 1.7x for MWT=3 compared to YARN.

Figure 4.4 and Table 4.1 reveal that *simple queues at worker nodes, even if bounded, negatively impact job completion times most of the time.* Only in a small number of cases, for some homogeneous workloads, we saw improvement in JCT just by bounding queue lengths. However, as we show in Table 4.1 and later in subsection 4.7.3, queue bounding coupled with task prioritization brings significant JCT gains.

4.7.2 Task Placement at Queues

We now compare different task placement strategies. We use our two strategies, namely queue length-based and queue wait time-based placement (see section 3.2), as well as a random placement strategy that randomly assigns tasks to nodes. We use a fixed queue size of six slots with task prioritization disabled. Job completion times for these runs are shown in Figure 4.6. As expected, the placement that is based on queue wait time outperforms the rest of the strategies, since it uses richer information about the status of the queues. In particular, it improves median job completion

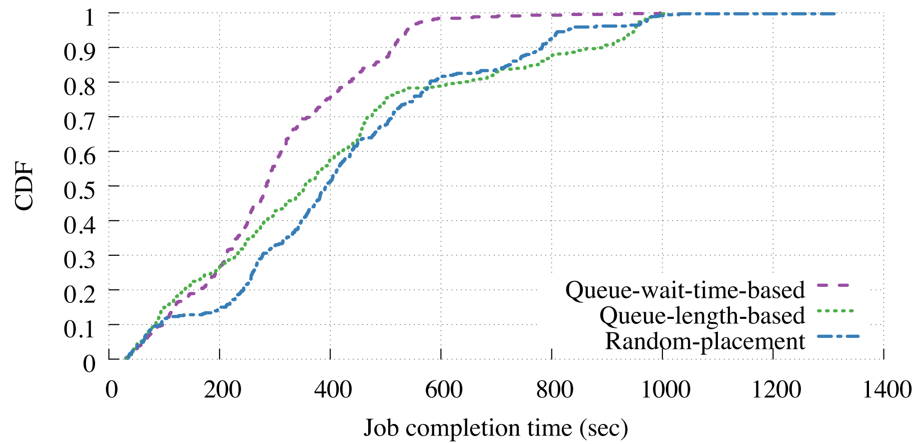

Figure 4.6: Job completion time for the GridMix-MS workload with different RM task placement policies.

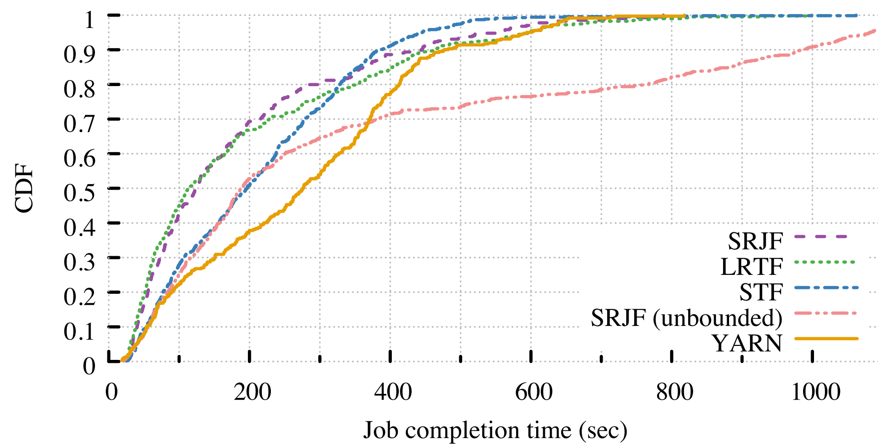

Figure 4.7: Job completion time for GridMix-MS workload with different task prioritization algorithms.

time by 1.2x when compared to the queue length-based and by 1.4x to the random strategy. Also note that the random placement has a significantly longer tail than our two strategies. Therefore, in all our experiments we use the queue wait time-based placement.

4.7.3 Task Prioritization at Worker Nodes

Figure 4.7 shows the job completion times (JCT) for our three task reordering algorithms (LRTF, SRJF, STF). We use a queue length of ten slots (unless otherwise stated) and no hard or relative

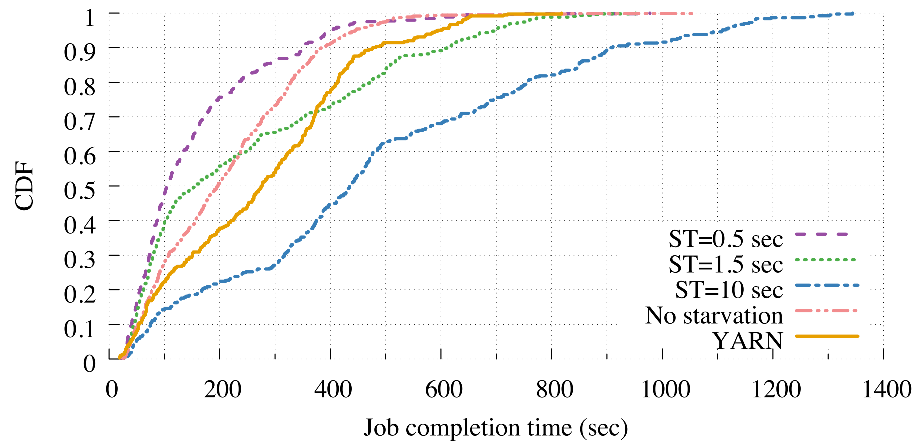

Figure 4.8: Job completion time for the GridMix-MS workload with different hard starvation thresholds (ST) and STF prioritization.

starvation thresholds.

The job-aware policies SRJF and LRTF perform the best when compared to YARN: 2.2x better median JCT for SRJF and 2.4x for LRTF. The non job-aware STF reordering policy performs 1.4x better than YARN. The difference in performance between STF and the other methods is that STF is more aggressive than others in attempting to fix head-of-line blocking issues, but can quickly lead to starvation issues (which are addressed later in this section). Thus, job progress is a much more reliable metric to use when reordering than the local metrics STF uses. Interestingly, for the GridMix workload LRTF performed better than SRJF (most probably due to the predictability of the synthetic workload). However, in the real Hive-MS workload, SRJF worked best.

In the same figure, we have included a run with SRJF prioritization and no queue bounding (marked “unbounded”). This run shows that with queue bounding disabled, task prioritization improves the lower percentiles of JCT, but negatively impacts the higher ones. Based also on the results of Table 4.1, it becomes clear that *combining task prioritization with queue bounding is required to get the best results in terms of JCT*.

Starvation Threshold We perform various runs to study the impact our starvation thresholds (see section 3.3) have on the performance of Yaq-c. The hard starvation threshold (ST) and relative starvation threshold (ST_r) both provide the ability to limit how long a task is starved. We empirically

found ST_r to provide less benefit in decreasing overall job completion time (JCT) when compared to the effects of ST. The results we present here showcase the effect of various hard starvation limits for the STF reordering policy, which benefits the most from the starvation parameter (given it is not job-aware as we discussed above). Figure 4.8 shows JCT with the GridMix-MS workload using STF reordering, a fixed queue size of ten slots, and various ST values. First, we observe that STF is sensitive to the ST value that is used. A value of 0.5 sec, which marks tasks as starved early, essentially falling back to the EJF strategy, works best for this synthetic workload with tasks of each job being relatively homogeneous. High values (ST=10 sec) are detrimental, whereas a value of 1.5 sec improves JCT for some of the jobs. Our experiments also revealed that SRJF and LRTF reordering are less sensitive to different ST values and that relatively higher values can give better results. Being job-aware, these strategies already prioritize the execution of starved straggler tasks. For instance, an ST value of 10 sec worked best on the more realistic Hive-MS production workload with SRJF. This also suggests that the ST value should be calibrated based on the characteristics of the workloads and the used strategy.

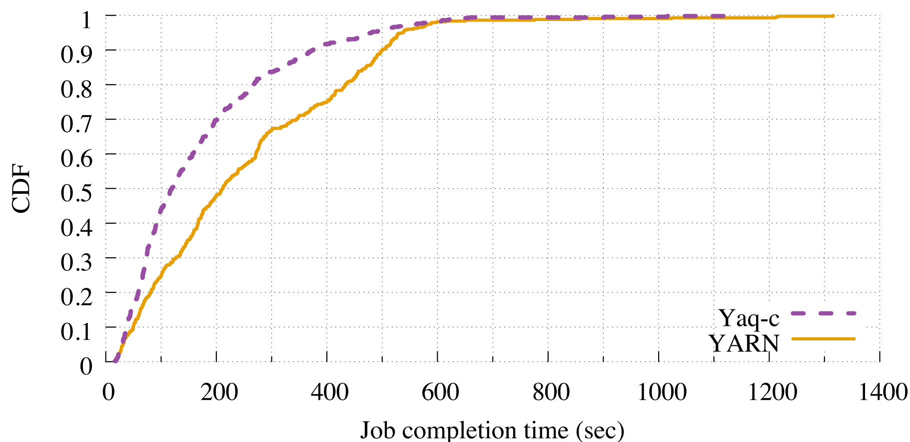

Figure 4.9: Job completion time for the heavy-tailed workload that is based on GridMix-MS.

4.7.4 Heavy-tailed Workload

The task durations of the heterogeneous workloads we have used so far (GridMix-MS and Hive-MS) follow an exponential distribution. In order to assess the impact of our techniques on workloads

with different characteristics, we modified GridMix-MS so that its task durations follow a heavy-tailed distribution. Specifically, we increased the duration of the longest 20% tasks by 500 secs. We use Yaq-c with a queue length of ten slots and the SRJF reordering strategy. Figure 4.9 show the gains in JCT that Yaq-c yields for this heavy-tailed workload. In particular, it improves median job completion time by 1.8x when compared with YARN.

4.8 Related Work

Our focus in Yaq is on the effective management of local task queues at worker nodes, and, as such, is complementary to extensive prior work in centralized [25, 30, 58–60], distributed [13, 19, 44, 49], and hybrid [18, 35] cluster schedulers. We covered many aspects of these systems in previous sections (particularly, in chapter 2), and complement our discussion here.

Local queues exist by necessity in distributed schedulers, such as Sparrow [44], Tarcil [19], Hawk [18], Apollo [13], and Mercury [35]. Sparrow and Hawk rely on the power of two choices balancing technique when placing tasks. Tarcil extends Sparrow's placement by adopting a dynamically adjusted sample size. On the other hand, in Yaq-d, similar to Apollo and Mercury, each scheduler uses information about the nodes' status to perform more educated placement decisions, which is crucial for heterogeneous workloads.

To the best of our knowledge, in all existing schedulers, whenever a running task finishes the next task selected for execution is mostly based on FIFO. Apollo acknowledges that queues can go beyond FIFO and be reordered, but does not explore this in depth. In contrast, we present the first extensive study of the impact of different queue management strategies in the cluster's performance.

While simple to implement, FIFO ordering can cause head-of-line blocking whenever task execution times differ significantly. This in turn impacts predictability of job execution times. To mitigate this issue, existing systems take extensive corrective mechanisms, such as duplicate scheduling [13], dynamically rebalancing queues across nodes [35], work stealing [18], and straggler mitigation [4, 23, 68]. Since head-of-line blocking issues are inherent to queuing systems, similar to these systems, Yaq also incorporates corrective mechanisms. Yaq goes beyond these systems

by avoiding these problems in the first place via careful placement of tasks to nodes, bounding of queues and prioritization of task execution, thus improving job completion times.

Our task prioritization strategies (see section 3.3) have commonalities with multiprocessor scheduling [5]. For instance, SRJF is similar to the Shortest Remaining Time First (SRTF) scheduling algorithm. However, unlike OS scheduling, SRJF relies on job progress information arriving from the RM/UM periodically, which can be stale. Moreover, in Yaq we can only perform local reordering of tasks once they have already been dispatched to a worker node.

Finally, our queue management techniques are related to the scheduling of packet flows in networks. The goal in network scheduling is to find a sweet spot between bandwidth utilization and flow completion time, which can be seen as related to cluster utilization and job completion time in cluster scheduling, respectively. Recent work like, PDQ [31] schedules flows based on earliest deadline first, pFabric [3] relies on remaining flow size, and DeTail [70] on application priorities. QJump [27] prioritizes packets based on flow classes, set by a network administrator.

Chapter 5

HyperDrive: Utility-Based Cluster Management

5.1 Motivation

In recent years many machine-learning frameworks have been introduced to help developers build and train machine learning models for solving different artificial intelligence tasks across supervised, unsupervised, and reinforcement learning domains [2, 12, 15, 16, 34, 65]. The task performance of trained models is very sensitive to many different adjustable parameters, called hyperparameters, which are configured prior to training a model. Examples of these hyperparameters include learning rate (in many models), number and size of hidden layers in a deep neural network, number of clusters in k-means clustering, and many more. Hyperparameter exploration searches across different configurations of a model, where each *configuration* represents a specific set of hyperparameter values. Its goal is to find good configurations that optimize the model performance (e.g., high accuracy, low loss, high reward) with affordable time and cost. This exploration involves two related problems: generating candidate configurations from the large space of hyperparameter settings, and actually scheduling and running these configurations.

Efficient hyperparameter exploration is of great importance to practitioners in order to improve model performance, reduce training time, and optimize resource usage. This is especially critical

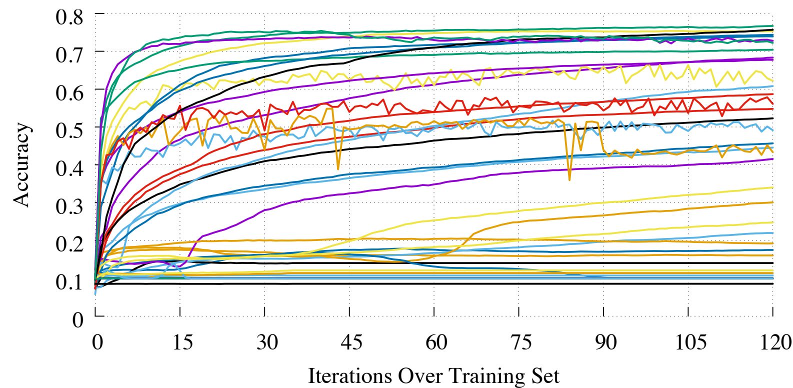

Figure 5.1: Performance of 50 randomly selected supervised-learning hyperparameter configurations.

when training modern deep and complex learning models with billions of parameters and millions of training samples on cloud resources. To reach a desired training target, efficient hyperparameter exploration means shorter training time, lower resource consumption, and thus lower training costs.

However, it is challenging to design effective scheduling frameworks and algorithms that efficiently explore hyperparameter values while obtaining high model performance and optimizing time and resource costs. The first key reason is the size of the hyperparameter search space and the expensive training process for each individual configuration. Figure 5.1 shows the model performance (task accuracy) of 50 configurations as a function of iterations over a training dataset for a moderate-size image classification application CIFAR-10 [40] (detailed experimental setup is presented in chapter 8). Each line in the plot represents a unique configuration. Here we present only 50 configurations in the plot for clear presentation. Each configuration needs to run about 120 iterations with each iteration taking about one minute. To fully explore only 50 configurations, we need over 4 days of computing. Commonly, models require exploring many more configurations to find high performing hyperparameters. For example, our CIFAR-10 model has 14 hyperparameters (such as learning rate, momentum, and weight decay) and most have a continuous range to explore, which results in hundreds or thousands (or more) possible configurations to explore. This problem is even worse when considering larger training models and datasets, for example, prior work has shown that a high-quality ImageNet22k image classification model can take up to ten days to train

to convergence using 62 machines [14]. Therefore exhaustive search is simply not practical.

A second reason is that, for many models in practice, only few configurations lead to high performance while a majority of configurations perform very poorly. The results in Figure 5.1, for example, show that only three configurations are able to exceed 75% accuracy (which is considered reasonable accuracy for this type of simple CIFAR-10 model that doesn't do any data augmentation and/or intensive preprocessing steps [1]), while the majority of configurations are not able to exceed 20% accuracy. Therefore, simple and popular approaches such as grid and random hyperparameter generation bet heavily on luck, which results in very inefficient discovery of high performing configurations under reasonable cost and time constraints.

Furthermore, optimizing hyperparameter tuning involves many other factors, such as incorporating different application domain goals (e.g., supervised, unsupervised, and reinforcement learning) and applicability to different DL/ML frameworks (e.g., Tensorflow, Caffe, and CNTK). It is challenging to design an effective framework to support such a wide range of usage scenarios.

Recent work [11, 33, 52] has moved beyond grid-based hyperparameter generation with adaptive techniques using Bayesian optimization, which assigns higher probability to areas of the hyperparameter space that contain likely-good or unknown configurations. These works do not, however, address how to run these configurations. For example, how long should each configuration run? Prior work [11, 33, 38, 52, 56] executes each configuration to the same maximum iteration (which can be a large number of iterations), typically ignoring the fact that some configurations could have shown their intrinsic value much earlier. As shown in Figure 5.1, with basic domain knowledge, one can quickly tell that many configurations do not learn at all, with accuracy similar to random (10% accuracy in this case), which can be identified within few training iterations and terminated early to save resources. In addition, should all running configurations take the same amount of resources? Clearly, that is not the best way to assign resources since the execution progress of the configurations could have offered many insights on how well the configurations are likely to perform, thus deserving different amounts of resources. This scheduling problem of how to effectively

[1] `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`

map different configurations across the time and space dimension of resources is largely unattended by prior work.

Our work focuses on designing an efficient scheduling algorithm and an effective system framework for hyperparameter exploration. Our scheduling algorithm, POP, dynamically classifies configurations into three categories: Promising, Opportunistic, and Poor, and uses these in two major ways. First, along the time dimension, we quickly identify and terminate poor configurations that are not learning, incorporating application-specific input from model owners. For example, classification tasks have known non-learning random performance values which can be used to prune jobs. Second, along the space dimension, we use an explore-exploit strategy. We classify promising configurations that are more likely to lead to high task accuracy and prioritize them with more resources while giving configurations that are still only *potentially* promising some chances to run – these are what we call opportunistic configurations. Unlike prior work [53] that uses instantaneous accuracy only to identify a fixed set of promising configurations, we incorporate the trajectory of full learning curves and leverage a probabilistic model to predict expected accuracy improvement over time as well as prediction confidence [20]. The classification and resource allocation between promising and opportunistic configurations is dynamic, adjusting the ratio of exploration and exploitation when we observe more predicted and measured results. At the early stage of training, when there is little history information and prediction confidence is typically low, allocating more resources for exploration helps training efficiency. Conversely, at later stages, confidence is higher, and allocating more resources for exploitation can yield higher rewards.

We also design a framework, HyperDrive, which serves as a generic framework for hyperparameter exploration. HyperDrive largely decouples the scheduling policy for candidate configurations from the type of model and/or framework. It provides an API that supports not only our POP scheduling algorithm, but also existing and new ones. It also supports different learning frameworks, such as Caffe [34], CNTK [65], and TensorFlow [2], and learning domains, such as supervised and reinforcement learning. Lastly, it supports model-owner-defined metrics and inputs to improve scheduling efficiency.

Chapter 6

Utility-Based Cluster Scheduling Design Principles

Hyperparameter exploration is challenging as there are many configurations but often much less time and fewer resources. For deep and complex models, it is common that only a small number of configuration choices lead to high quality results. Thus, significant amounts of time and resources could be wasted in searching for good configurations if considerable attention is not paid to search efficiency. To enable efficient model exploration and scheduling, we follow three key design principles: i. early identification and termination of *poor configurations* that are either not learning or learning very slowly; ii. early classification of *promising configurations* that are more likely to lead to high task performance among the remaining group of *opportunistic configurations*; iii. prioritized execution of promising configurations by devoting more resources to them without starving opportunistic configurations; striking a desired balance between *exploitation* of promising configurations and *exploration* of opportunistic configurations. Our search algorithm POP achieves these design principles by solving the following three challenges.

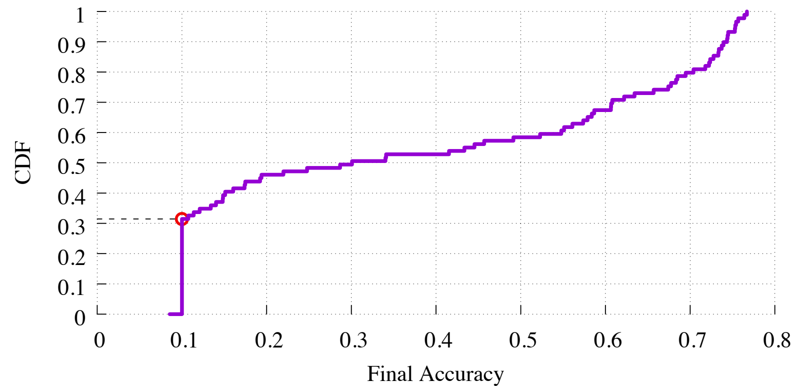

Figure 6.1: Final validation accuracy distribution of 90 randomly selected CIFAR-10 configurations.

6.1 Identify Poor Configurations Early

Early detection of configurations with poor learning performance can reduce wasted resources during model search. Figure 5.1 shows that in the search for high-quality *CIFAR-10* models a significant portion of possible model configurations either do not learn or learn very slowly during the entire training process. We present the final validation accuracy distribution of different configurations in Figure 6.1. The red circle shows the percentile of configurations that achieve below the random validation accuracy of 10%[1] We can see there are 32% of configurations with poor validation accuracy, i.e., at or below random validation accuracy. Such a significant amount of configurations with poor validation accuracy demonstrates the importance of identifying and terminating poor configurations as early as possible to reduce wasted time and resources.

An efficient way to identify *poor configurations* early is to incorporate domain knowledge from the model owner. For example, in many supervised-learning tasks it is common for poor hyperparameter values to result in models that only achieve "random" validation accuracy (which is defined by the task), an example of this can be seen in Figure 5.1, where the task is forced to choose a label from 10 categories, therefore many configurations only achieve random validation accuracy

[1] Random accuracy is defined as 10% here due to CIFAR-10 having 10 categories, thus a random guess yields a 10% chance of being correct.

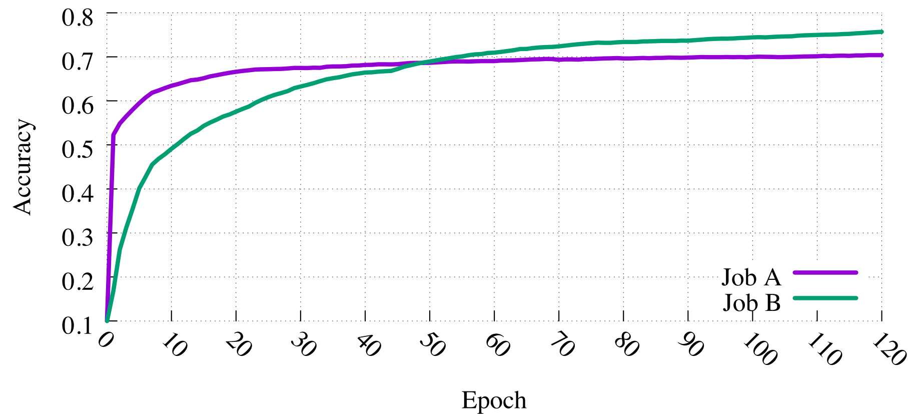

Figure 6.2: Final learning curves for two different configurations A and B.

around 10%. The search algorithm can incorporate this knowledge to improve search by terminating configurations early if they fail to escape this "random" validation accuracy threshold after a few iterations. Similarly, a user can incorporate early termination for many reinforcement-learning models due to a common "not learning" range which can be determined based on the environment being trained on. In addition it is common for reinforcement-learning tasks to also have unique "solved" conditions that can be incorporated into a search algorithm, for example a task may only be considered "solved" when it sustains a certain reward for some number of iterations.

6.2 Classify Promising Configurations Early and Judiciously

To classify *promising configuration* early and judiciously, one effective way is to develop an accurate methodology for predicting future task performance. There are three important questions to answer for developing an accurate prediction methodology.

a) Would the most recent performance alone be sufficient? As an example, Figure 6.2 shows full validation accuracy curves for two configurations A and B. At the early stage, i.e., before the 50th epoch, A's validation accuracy is higher than B's. However, the final validation accuracy of B is higher than A, thus B *overtakes* A. If we simply rely on the most recent performance, we will not discover that B is the most promising configuration until after the 50th epoch and thus waste a lot

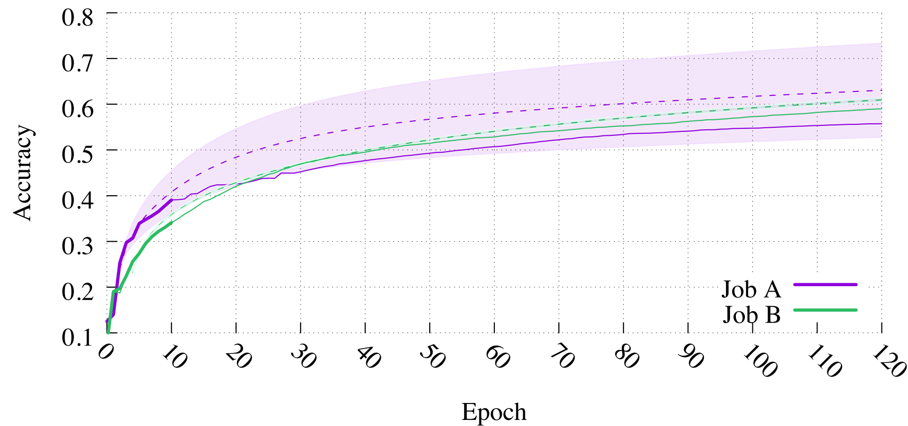

Figure 6.3: Job A has higher predicted accuracy but lower confidence, while Job B has better final accuracy and higher confidence.

of resources. We observe this overtake phenomenon sometimes can even be more pronounced than seen in Figure 6.2. Therefore, in order to classify *promising configuration* early, the most recent performance alone, as used in prior work [53], is not enough. In order to make effective predictions, we use a probabilistic model to predict expected future task performance by incorporating partial task performance history, i.e., the task's learning curve.

b) Would predicting expected future task performance

alone be sufficient? We answer the question by showing an example using Figure 6.3. The dotted lines show A and B's expected validation accuracy and the solid lines show the measured validation accuracy. The results indicate A's expected validation accuracy is higher than B at epoch 10 but with much larger variance and lower confidence than B (the shadow represents the confidence intervals). However, in the final validation accuracy, B is actually higher than A, which indicates expected future validation accuracy alone can be misleading and we need to assess the quality of the prediction. To quantify the prediction quality, we calculate the confidence of the prediction.

c) Would a static threshold be sufficient to decide a *promising configuration*? One way to classify promising configurations is by using a static threshold for the probability of achieving target task performance, e.g., if the probability is higher than the threshold, the configuration is promising. However, the problem with this approach is if the threshold is too high, it becomes difficult to

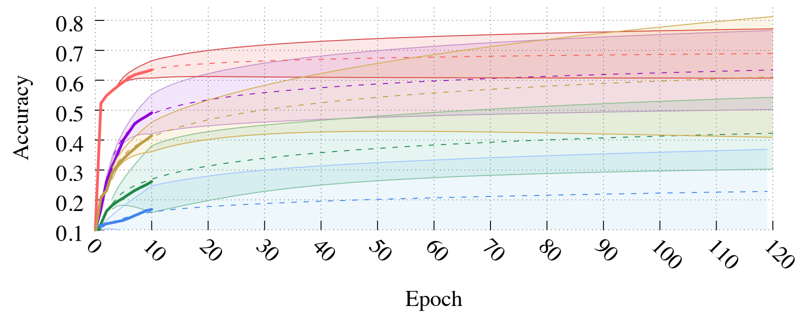

Figure 6.4: Validation accuracy predictions at epoch 10.

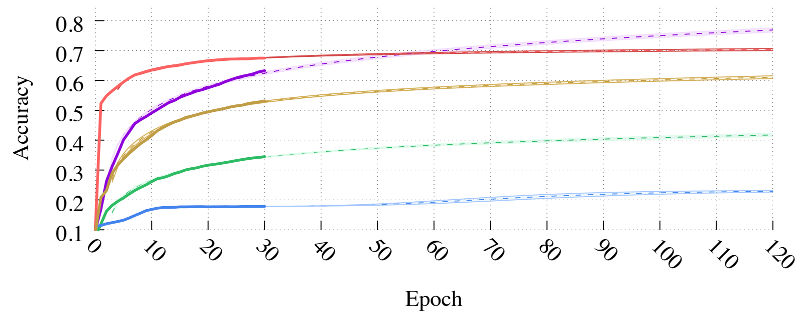

Figure 6.5: Validation accuracy predictions at epoch 30.

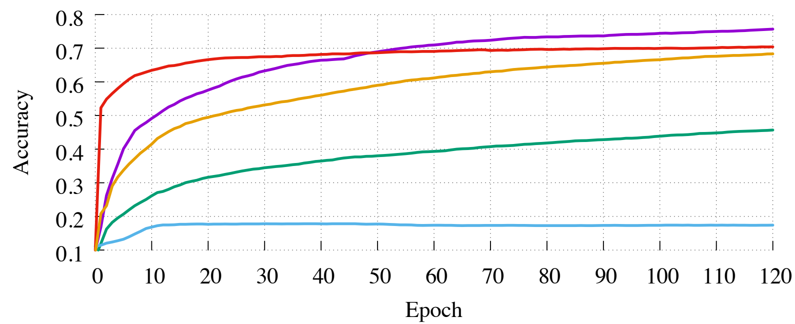

Figure 6.6: Final validation accuracy at epoch 120.

identify promising configurations early in the training process. On the other hand, if the threshold is too low, we may classify too many configurations as promising and results in an ineffective way to allocate resources. Therefore, when determining the threshold, we need to take into consideration both the characteristics of the model and the available resources.

6.3 Resource Allocation Between Promising and Opportunistic Configurations

The key insight here is that static resource allocation between promising and opportunistic configurations is insufficient as configurations can change status between promising, opportunistic, and poor over time. Figures 6.4, 6.5, 6.6 give examples of prediction confidence at three different stages during training. For example, at the beginning stage Figure 6.4, there is little trajectory information and thus low confidence to differentiate configurations: all active configurations are classified as opportunistic and all resources are designated opportunistic. As training progresses, more promising configurations emerge thus we allocate more resources for promising configurations. At later stages, it is possible that one or several configurations have very high confidence to achieve their target, therefore we can allocate resources to them in a much more aggressive way or even use an "all-in" strategy. Therefore, the configuration classification and resource allocation should be coordinated based on the progress of training.

6.4 Design Overview of HyperDrive

In designing the POP scheduling algorithm, two things became clear: first, the concerns of the policy are largely independent of the exact learning domain or framework, provided the scheduler can extract the right information from the tasks; second, to efficiently schedule the learning tasks, we needed a slightly richer interface than that of traditional task schedulers such as YARN or Spark. For example, we needed the ability to suspend and resume tasks to effect resource allocation, and to convey to the policy model-owner-defined metrics.

Our HyperDrive framework addresses these observations, and is a step towards providing a separation between hyperparameter search algorithms and their runtime environment.

6.4.1 Design Considerations

We designed HyperDrive with the following goals in mind:

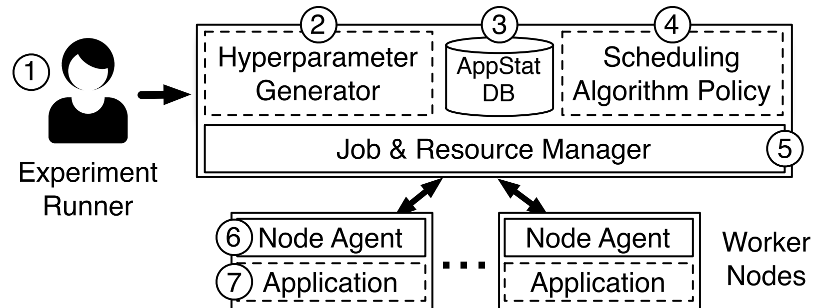

Figure 6.7: HyperDrive architecture

Support and enable reuse of existing and future search and scheduling algorithms. As there will be new or customized hyperparameter optimization methodologies, the scheduling framework shall be flexible enough to allow users to swap in and out different search and scheduling algorithms.

Monitor and report job status to support dynamic resource adjustment and early termination. To support judicious scheduling decisions, such as dynamic resource adjustment and early termination, the framework should be able to monitor and report current and history job status.

Support different learning domains by allowing inputs from model owners. The scheduling framework should support different learning domains (e.g., supervised, unsupervised, reinforcement learning) by allowing model owners to specify domain specific requirements.

Support different learning frameworks. There are many different learning frameworks in use today, such as CNTK, TensorFlow, Theano, Caffe, MXNet, etc. The scheduling framework should be learning framework agnostic, i.e., not bound to a specific one.

6.4.2 HyperDrive Framework

Figure 6.7 shows HyperDrive architecture, which is described below. Numbers in circles correspond to the components in the figure.

Job and Resource Management The core job and resource management components of HyperDrive (⑤) provide the basic ability of executing jobs on remote machines.

The Resource Management (RM) component is responsible for keeping track of currently allocated and idle resources (e.g., machines, GPUs). We leave its description short for brevity and its

simplicity. However, if executing HyperDrive in a cloud environment this piece is customized for the specific environment (e.g., reserve an Azure/AWS instance). The RM provides a simple API to other components:

- $reserveIdleMachine() \rightarrow machineId$
- $releaseMachine(machineId)$

The Job Manager (JM) provides the ability to start, resume, suspend, and terminate jobs on specific machines obtained from the RM. It keeps track of each job's state based on the actions performed on it. The JM provides the following API to other components:

- $getIdleJob() \rightarrow jobID$
- $startJob(jobID, machineID)$
- $resumeJob(jobID, machineID)$
- $suspendJob(jobID, machineID)$
- $terminateJob(jobID, machineID)$
- $labelJob(jobID, priority)$

Suspend and resume support is used to enable flexible scheduling of jobs, which means that the framework must be able to train a model for an unspecified amount of time, suspend training, and then resume training later on any machine associated with the experiment. Suspend and resume requires that training state is saved and synchronized with the AppStat database (③), which allows any machine to recieve the state and resume training. The JM also provides the ability to label a job with a priority value, which the JM uses to order idle jobs. Priority ordering is especially important when adding a suspended job to the list of idle jobs. If no priority is given then idle jobs are ordered according to FIFO order.

Node Agent The Node Agent (⑥) is daemon running on a worker machine responsible for job execution and acting as an intermediary between the HyperDrive scheduler and the training application (⑦). All Job Manager calls from HyperDrive that deal with job execution are received and

executed by a Node Agent. In addition all application statistics reported by the training application are sent to its local Node Agent and then forwarded to the HyperDrive scheduler.

Scheduling Algorithm Policy A user-provided Scheduling Algorithm Policy (SAP) (④) is written in an imperative style using the following three HyperDrive *up-call* events:

- $AllocateJobs()$
- $ApplicationStat(jobEvent)$
- $OnIterationFinish(jobEvent)$

$AllocationJobs$ is triggered on detection of an idle resource to allow the SAP to schedule a new job on that resource. $ApplicationStat$ is triggered on receiving application stats (e.g., accuracy) from the training job to enable the SAP to store or process the data as appropriate. Lastly, $OnIterationFinish$ is triggered when a training iteration finishes to allow the SAP to decide whether to continue, suspend, or terminate the job, or collect additional statistics (e.g., iteration timings). We find that these simple scheduling primitives allow us to write a diverse collection of SAPs that cover some prior work and our own scheduling algorithm as described in chapter 6.

A SAP is notified when a job finishes an iteration. Then it makes a decision whether to continue training the job or terminate/suspend the job. By default, HyperDrive uses a schedule-as-it-goes approach to maximize resource usage since configurations with short epoch durations do not need to wait for those with long durations. HyperDrive also supports barrier-like epoch scheduling, which some SAPs may prefer as it can help explore job configurations in a breadth-first-style (i.e., executing many jobs for a short period of time in each round). Barrier-like epoch scheduling can be achieved by allowing the SAP to suspend jobs at every epoch boundary.

Default SAP The default SAP simply greedily allocates idle jobs to idle machines, which is implemented by starting as many idle jobs (via $startJob$) as there are idle machines. This policy ignores all application statistics and iteration finish up-calls, but provides a simple base for more advanced SAPs.

Hyperparameter Generator The Hyperparameter Generator (②, HG) is responsible for generating specific parameter values within ranges specified by the experiment runner. The generator

implementation is pluggable as long as it provides the following API:

- $createJob() \rightarrow (jobID, hyperparameters)$
- $reportFinalPerformance(jobID, performance)$

We consider the use of several different HG techniques, which are built separate from HyperDrive itself. Along with more complicated approaches we have built simple random and grid search techniques as HGs, where a user-provides the parameter names and search ranges. The HG then selects new random or grid values upon each call to $createJob$. In these approaches the $reportFinalPerformance$ call is not used.

Adapative techniques (e.g., Bayesian optimization) are popular alternatives to random and grid search and used in in frameworks like HyperOpt [38], Auto-WEKA [56], Spearmint [52], and GPyOpt [10]. These frameworks generate new hyperparameter values based on the observed performance of previous values. These type of approaches can be plugged into HyperDrive with the use of a shim that exposes the HG API.

AppStat Database The application statistics database (AppStatDB ③) is used to store and retrieve model-generated application statistics such as performance stats (e.g., accuracy, reward), epoch duration, etc. In addition the AppStatDB stores model state used to enable suspend and resume training across machines. The AppStatDB is used to sharea state between the SAP, Hyperparameter Generator, and training job itself.

Experiment Runner (Client) The Experiment Runner (①) is responsible for specifying the following items when running an experiment with HyperDrive:

- Search Algorithm Policy to use (with any SAP specific parameters)
- Hyperparameter generation technique along with parameter names and search ranges
- Model training files to run on remote machines
- Total number of machines

Chapter 7

Scheduling Algorithm and Framework Implementation

The search for high-quality models, from a candidate set of model configurations, typically involves multiple iterations of training each model configuration with a training dataset and evaluating model performance against a validation dataset. The objective of POP is to improve the efficiency of discovering high performing model configurations: minimize the time to find a configuration satisfying a target performance. It can also be used for finding configurations with the best performance within a time budget, which is a corresponding dual problem. To achieve efficient search and scheduling, we need to promptly and accurately identify configurations that are more likely to result in high-quality models i.e., *promising configurations* (section 7.1). In addition, we need an efficient resource allocation strategy that prioritizes promising configurations. Here we develop an infused methodology incorporating both configuration classification and scheduling (section 7.2).

7.1 Configuration Classification

As discussed in chapter 6, in order to classify configurations, POP needs to consider both the expected final task performance and the quality of prediction based on performance history information. Recall that the objective is to minimize the time to find a configuration that achieves a

target performance. Therefore, at any point of time in training, the configurations with the smallest expected remaining time to achieve the target performance and with high prediction quality are considered promising configurations. To classify configurations accurately and promptly, an accurate estimation model of the expected remaining time and a methodology to evaluate the prediction quality is key.

Expected Remaining Time Estimation

We develop a probabilistic approach for estimating the expected remaining time for a given configuration.

In many learning domains it is common to periodically evaluate a model's performance (e.g., validation accuracy in supervised-learning). The frequency at which a model's performance is evaluated is often at the end of a training epoch, so to compute the expected remaining time, we can first compute the expected number of remaining epochs and then multiply with the average epoch duration[1]. The idea is to compute the probability of achieving the target performance at future epochs and then use a probability mass function to estimate the expected number of remaining epochs.

Problem formulation: We define the problem as predicting the expected remaining time for a given configuration to achieve the target accuracy. The following parameters are required from users as input parameters[2]:

- T_{max}: the maximum experiment time a user can tolerate;
- y_{target}: a target model performance;

These two parameters are based on the user's domain knowledge, therefore values provided by experts should ideally lower estimation overheads and improve search efficiency compared to values provided by beginners. As we first do prediction based on epochs, we compute the maximum number of remaining epochs M_i for a given configuration i as $M_i = (T_{max} - T_{pass})/Epoch_i$, where T_{pass}

[1] An epoch represents training over an entire training data set once. Epoch durations are assumed to be roughly constant, see section 9.1 for more details.

[2] See section 9.1 for a discussion on these user parameters.

is the measured time duration that has passed from the beginning of the experiment, and $Epoch_i$ is the measured average epoch duration. We define $p1, p2, ..., p_m, ..., P_M$ as the probability that the target performance can be reached in $1st, 2nd, ..., Mth$ epoch and p as the prediction confidence, which is defined as the probability that a configuration can achieve the target performance within T_{max}, i.e., $p = p1 + p2 + ... + p_m + ... + P_M$. The prediction model output is the expected remaining time ERT_i for configuration i to achieve target accuracy y_{target}.

Model performance prediction: To predict future model performance, we rely on the configuration's learning curve. More specifically, we compute the probability $P(m)_i$ of configuration i reaching a model performance y after epoch m in the future based on the observed validation performance history of the configuration, as follows:

$$P(m)_i = P(y(m)_i \geq y | y(1 : m-1)_i), \tag{7.1}$$

where $y(m)_i$ is the predicted performance after epoch m for configuration i and $y(1 : m-1)_i)$ represents the observed performance of configuration i from epoch 1 to $m-1$. We leverage a probabilistic learning curve model proposed in prior work [20] to compute $P(y(m)_i \geq y | y(1 : m-1)_i)$.

The learning curve prediction model we use relies on a weighted combination of 11 different parametric models (e.g., vapor pressure, Weibull, Janoschek) and uses Markov Chain Monte Carlo (MCMC) inference to predict possible values of these weights based on the observed partial performance curve. This probabilistic model is then used to compute the probability $P(m)_i$ of each configuration periodically online, which allows the scheduling policy to see a global view of performance across all active configurations. Due to the non-deterministic nature of MCMC inference, we define a prediction accuracy PA to be the standard deviation across all MCMC samples. Further discussion of our implementation and optimization of the learning curve prediction model is discussed later in section 7.5.

Modeling expected remaining time and prediction confidence: To model the expected remaining training epochs x_i for configuration i to achieve the target performance y_{target}, we compute

the probability that the target performance can be reached at the $1st, 2nd, ..., mth, ..., Mth$ epoch respectively[3]. According to the definition of the probability mass function defined based on accumulative distribution, we have:

$p1 = P(y(1)_i \geq y_{target})$,

$p2 = P(y(2)_i \geq y_{target}) - P(y(1)_i \geq y_{target})$,

...

$p_m = P(y(m)_i \geq y_{target}) - P(y(m-1)_i \geq y_{target})$,

...

$p_M = P(y(M)_i \geq y_{target}) - P(y(M-1)_i \geq y_{target})$.

Thus the expected number of remaining epochs x_i for configuration i can be estimated as:

$$x_i = 1 * p1 + 2 * p2 + ... + m * p_m + ... + M * p_M \tag{7.2}$$

Therefore the expected remaining training time ERT_i for configuration i is:

$$\begin{aligned} ERT_i &= x_i * Epoch_i \\ &= (1 * p1 + 2 * p2 + ... + m * p_m + ... \\ &+ M * p_M) * Epoch_i \end{aligned} \tag{7.3}$$

Ideally, the probability $p1, p2, ..., p_m, ..., P_M$ should sum to 100% (i.e., $\sum_{m=1}^{M} (p_m) = 1$). But in reality, we do not need to sum further if the expected remaining training time is larger than the maximum experiment time duration that user can tolerate, i.e., $ERT_i > T_{max} - T_{pass}$. In other words, we stop summing further for p_m and set $ERT_i = T_{max} - T_{pass}$ since the search algorithm will not run further than $T_{max} - T_{pass}$. Therefore, the probability $p1, p2, ..., p_m, ..., P_M$ may not sum up to 100% (i.e., $\sum_{m=1}^{M} (p_m) \leq 1$). Here we define the probability sum as the prediction confidence p as the higher the probability sum, the more certain the expected remaining training time[4].

[3]Note M is actually M_i because it is configuration-specific. We omit i here for ease of presentation.

[4]Note p is actually p_i because it is configuration-specific. We omit i here for ease of presentation.

Classify configurations: We define p_{thred} as a threshold for prediction confidence of classifying promising configurations: if $p \geq p_{thred}$, then the configuration is classified as a *promising configuration*, otherwise the configuration is classified as an *opportunistic configuration* or *poor configuration*. To distinguish between *opportunistic configuration* and *poor configuration*, we rely on the domain knowledge as explained in chapter 6. The remaining question is how to determine the classification threshold p_{thred}? As explained in chapter 6, a static threshold is insufficient and must consider available resources to determine the threshold value. Next, we develop an infused methodology for determining the threshold and making judicious scheduling decisions.

7.2 Infused Classification & Scheduling Methodology

Ideally, if we could perfectly predict future model performance and expected remaining time of model configurations, we could allocate all resources to the most promising configuration(s) (i.e., with shortest expected remaining time). However, in practice, since predication cannot be 100% accurate, we need to allocate resources based on the prediction quality as well as the available resources. The proposed search algorithm employs both an exploration and exploitation approach for resource allocation. We allocate dedicated resources to exploit promising configurations as they are more likely to produce high quality results; we also reserve resources for exploring the opportunistic configurations as when more information is available, i.e., after more epochs of execution, they may become promising. Therefore, the available resources are divided into two pools accordingly: a *promising resource pool* and an *opportunistic resource pool* (we do not allocate resources for *poor configurations*). We dynamically adjust the resource division based on the computed prediction confidence and measured prediction accuracy. In other words, during training, we adjust the ratio of resources dedicated for exploitation versus exploration as we observe more predicted and measured results.

Assume S is the total number of slots (e.g., machines, GPUs), which is typically much smaller than the total number of configurations. Let $N_{satisfying}(p)$ denote the number of configurations with confidence p that can achieve the target performance within the maximum experiment time that the

user can tolerate, or equivalent to $\{i|ERT_i(p) \leq T_{MAX}\}$, where $ERT_i(p)$ is the expected remaining training time for configuration i to achieve the target performance with confidence p (an extended definition of the expected remaining training time ERT_i). Naturally, a large $N_{satisfying}(p)$ value under the same confidence p corresponds to a large number of promising configurations. Also, high values of confidence p typically results in small values of $N_{satisfying}(p)$.

To decide the effective number of slots $S_{effective}$ for promising configurations, we look at the problem from two angles, their *desired* number of slots $S_{desired}$ and the *deserved* number of slots $S_{deserved}$. For any given confidence p, we consider those configurations satisfying the confidence as promising, i.e., the number of promising configurations is $N_{satisfying}(p)$. Assume each promising configuration gets a dedicated number of slots k. For example, if a slot represents a machine, a sequential execution of a configuration has $k = 1$ The desired number of slots for promising configurations $S_{desired}(p)$ is equal to $N_{satisfying}(p) \times k$. On the other hand, the total number of slots is limited by resource availability, and the number of slots promising jobs *deserve* is related to the confidence p — the higher the confidence, the more resources they shall get. We calculate the desired number of slots as $S_{deserved}(p) = S \times p$. The actual resources that promising jobs shall receive must be both desired and deserved, and thus $S_{effective}(p) = \min(S_{desired}(p), S_{deserved}(p))$.

Among all p values, we choose the one that maximizes $S_{effective}(p)$, i.e., the number of slots for promising configurations is equal to $S_{promising} = argmax_p(S_{effective}(p))$. These slots are assigned to run promising configurations with dedicated resources. The remaining slots are allocated to the *opportunistic resource pool*, where the resources are equally shared between opportunistic configurations, e.g., in a round robin manner.

Figure 7.1 and Figure 7.2 show the number of desired slots and deserved slots under different prediction confidence values p. Figure 7.1 shows a snapshot taken in the early stage of an experiment (after about 20 min) when most of the p values are very small due to limited history information available. Figure 7.2 is a snapshot taken at a later stage of the experiment (after almost 2 hrs). From both figures, we can see that: (1) $S_{desired}(p)$ is a monotonically non-increasing function of p, since when p increases, $N_{satisfying}(p)$ will not increase and can only decrease. (2) $S_{deserved}(p)$ is a monotonically increasing function of p, since higher p deserves more resources. The cross point of

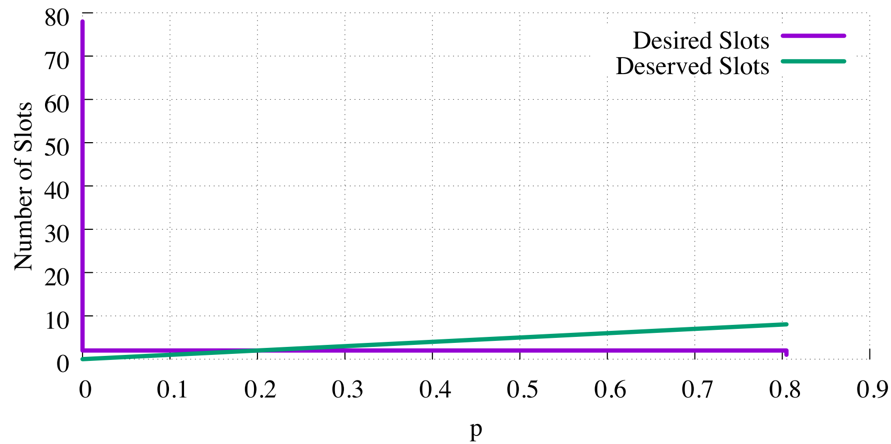

Figure 7.1: Desired slots are low early on due to low confidence.

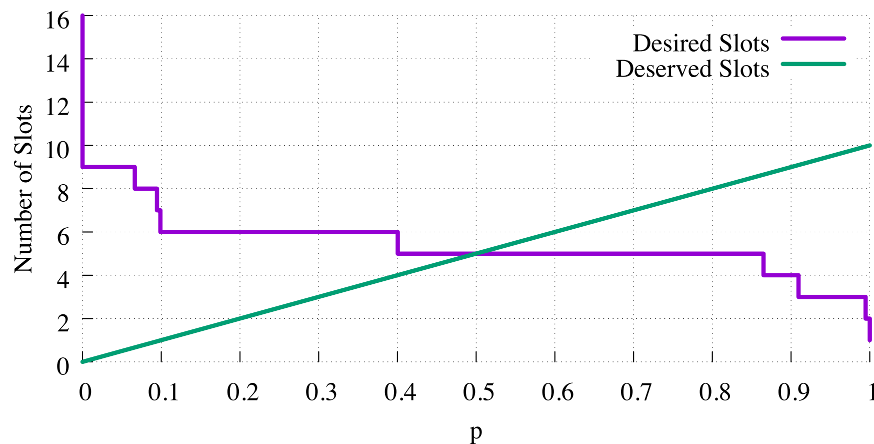

Figure 7.2: Desired slots are high later on due to higher confidence.

the desired slot and deserved slot curves maximizes $S_{effective}$, which corresponds to the number of slots given to promising configurations $S_{promising}$.

To further understand how our resource allocations change over time, consider Figure 7.3 which shows how the ratio of resources allocated for exploitation (the *promising resource pool*) versus resources for exploration (the *opportunistic resource pool*) change over the experiment's lifetime. It is clear that at the early stage a higher share of resources are used for exploration, however later the share of exploitation resources increase significantly as we improve our overall prediction quality.

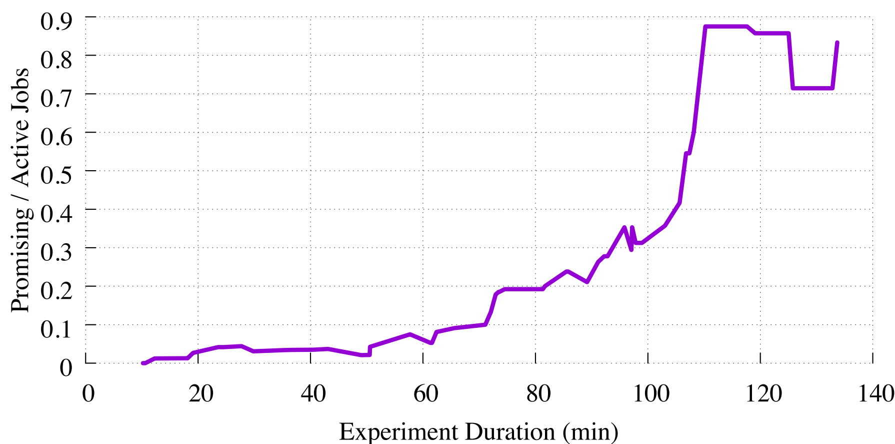

Figure 7.3: Ratio of promising slots increases over time.

7.3 Implementation of HyperDrive

We implemented a HyperDrive prototype based on the components described in section 6.4 in Python. All communication between the scheduler, node agents, and applications is done via GRPC [1]. We implemented two HyperDrive application libraries in Python and C++ that we use to support Theano, Keras, TensorFlow, and Caffe.

7.4 Suspend & Resume Support

In order for suspend and resume to work correctly we must either have support from the training framework itself to save all necessary training state to disk. Many frameworks have the ability to do something like this, particularly Caffe takes *snapshots* by default periodically during training or when it receives a kill signal. For our supervised-learning evaluation and all Caffe training jobs we use their snapshot and restore features to implement suspend and resume in HyperDrive. However, Caffe is less flexible in terms of the types of learning tasks it is able to evaluate when compared to other frameworks like Theano, Keras, and TensorFlow (e.g., reinforcement learning problems cannot be solved with Caffe). By adding additional flexibility to the training framework, users are able to write arbitrary Python-based models that take advantage of the powerful tools that frameworks like TensorFlow provide along with their own custom solution on top of them.

The problem that arises with this approach is that a user's model may contain numerous pieces of state outside the TensorFlow frameworks view. If all model state is maintained by the framework then often the framework's check-pointing mechanism will suffice. If however, the model writer maintains some of their own model state then the framework's check-pointing mechanism will not be sufficient. In order to cover additional model state scenarios, we take a different approach to suspend/resume for non-Caffe training jobs. With HyperDrive we instead utilize CRIU [5] to snapshot and resume the entire training job process's state. In practice we have found that this approach performs well and generalizes easily to other training frameworks and models.

7.5 Learning Curve Prediction

We implemented the learning curve prediction model by adapting a public implementation [6] of the model from [20]. The overhead of running the unmodified learning curve prediction model for a single learning curve can time consuming (several minutes). We identify three optimizations for performing parallel learning curve prediction in HyperDrive.

Reduce total MCMC samples. At its core the learning curve prediction module uses a computationally expensive Markov Chain Monte Carlo (MCMC) inference technique to predict future training performance. In order to reduce the total time to create the learning curve prediction model we reduced the total number of MCMC samples from 250,000 (nwalkers=100, nsamples=2500) to 70,000 (nwalkers=100, nsamples=700). This reduced our learning curve prediction time by over 2x without significant degradation in our policy's performance.

Distributed Curve Prediction. A simple implementation of HyperDrive would run all learning curve prediction at the central scheduler. However, this approach does not scale as the number of Node Agents increases since training jobs may require simultaneous curve predictions. Instead we push the learning curve prediction to the Node Agents. The Node Agents keep track of the curve history for each job they are responsible for and report to the central scheduler the results of

[5] https://criu.org/

[6] https://github.com/automl/pylearningcurvepredictor

a prediction. If a training job is suspended and resumed on different machines the learning curve history is sent to the new Node Agent when the job is resumed.

Overlap training and prediction. A simple prediction implementation would block training while the prediction is computed. Instead, as soon as the Node Agent detects that prediction should be started it does so in parallel to training. We have found, and our evaluation shows, that the end-to-end performance gains outweigh any slowdown that the training may experience due to resource contention. Although a similar approach was taken by [20], resource contention was not an issue since training was done exclusively on GPUs and prediction on CPUs. Our evaluation covers both CPU and GPU-based training using this strategy.

7.6 Scheduling Policies

We now describe how we use HyperDrive to implement three scheduling policies used in our evaluation: the POP policy from chapter 7 and two state-of-the-art policies from prior work: a bandit allocation policy from [53], and an early termination policy from [20].

POP We implement the POP algorithm as described in chapter 7. When $OnIterationFinish$ is called the policy checks to see if the current iteration (n) is on an evaluation boundary (b), if so we perform several steps. We first compute the expected number of iterations the job has remaining (k) Then compute our p value for the job as described in chapter 7. In order to calculate our desired and deserved slots we compute the tail distribution across all currently active (non-terminated) job's p values. Then compute our dynamic $p_{threshold}$. Then compare our threshold to all active jobs and determine if they are in our *promising resource pool* or not, we label each promising job (using $labelJob$) with a priority value of p. Lastly, if the job is opportunistic we suspend it and start a new job.

We now discuss domain and task-level knowledge we incorporate into POP to prune poor configurations. Before computing any learning prediction we first check to see if the job's performance has passed a user-defined *kill-threshold* based on the specific learning task. For example, in the CIFAR-10 task (discussed in section 8.1) it is known that random non-learning validation accuracy

is 10%, therefore we set the kill-threshold to a value slightly over random accuracy at 15%. In our LunarLander reinforcement-learning task (discussed in section 8.1) we know that non-learning performance is -100 therefore we set our kill-threshold to -100. In addition, in order to prune off jobs that are unlikely to achieve our target, we compare a job's p value against a lower-bound threshold, if it is less than 0.05 we terminate it. Lastly, we set b to 10 for supervised-learning and to 2,000 for reinforcement-learning.

Bandit Our Bandit policy is based on the action elimination algorithm [22] used by TuPAQ [53] in their bandit allocation strategy. In order to implement the Bandit policy we extend the Default SAP described in section 6.4. Model performance stats are sent to the policy every epoch, the SAP keeps track of the global best model performance ($globalBest$) along with the best model performance per job ($jobBest$). When $OnIterationFinish$ is called the SAP checks to see if the current iteration is on an evaluation boundary (b), if so it checks if $jobBest * (1 + \epsilon) > globalBest$. If true, the job continues training, if false the policy terminates the job. Based on prior work [53], ϵ is set to 0.50 and b is set to 10 for supervised-learning. Prior work focused on supervised-learning, therefore we have no guidance on setting an evaluation boundary for reinforcement-learning. Thus, we use the same value as our POP policy (i.e., 2,000 iterations).

EarlyTerm The EarlyTerm policy is a parallel version of prior work [20] that introduced the learning curve prediction model used in our POP policy, we use the same optimizations here as described in section 7.5. Like Bandit, we extend the default SAP. The EarlyTerm policy implements the "predictive termination criterion" described in [20]. Model performance stats are sent to the policy where it keeps track of the full history of performance across each job, along with $\hat{y}$ which is the global best model performance seen. When $OnIterationFinish$ is called the policy checks if the current iteration (n) is on an evaluation boundary (b), if so it computes $p_{val} = P(y_m \geq \hat{y} | y_{1:n})$ using its probabilistic model. If $p_{val} < \delta$ then the job is immediately terminated. The value of m is set to the max epoch set for the training jobs. We use the same b value of 30 and δ to 0.05 as [20]. Similar to the Bandit policy, prior work provides no guidance on b values for reinforcement-learning, thus we use the same value as our POP policy (i.e., 2,000 iterations).

Chapter 8

Evaluation of HyperDrive

We evaluate the effectiveness of our proposed POP scheduling algorithm and HyperDrive framework in two different domains: supervised-learning (section 8.2) and reinforcement-learning (section 8.3).

8.1 Experimental Setup

Scheduling Policies. In each learning domain, we compare POP against three baseline approaches: (1) Default, (2) Bandit, and (3) EarlyTerm. The Default policy (see section 6.4) schedules jobs greedily on idle machines and runs them until completion (i.e., a max number of epochs). Bandit and EarlyTerm are based on prior work and their implementation is described in section 7.6.

Workloads. For supervised learning, we use a popular image classification task, CIFAR-10 [40], that classifies 32x32 images into 10 categories (e.g., cat, truck). We use a convolutional neural network (CNN) based on the *layers-18pct* configuration from Krizhevsky's cuda-convnet [39]. Even though this model does not have state-of-the-art accuracy, it is a popular version coming with Caffe. State-of-the-art models often employ data augmentation and/or intensive preprocessing steps, which are orthogonal to hyperparameter exploration that our work focuses on. We follow the standard approach of training on 50k images and evaluating model performance on a validation dataset of 10k images.

For reinforcement learning, we use a model for a popular task from the OpenAI Gym [37] called "LunarLander". LunarLander comes from a game where an agent has control over a lander to do four discrete actions: do nothing, fire left engine, fire main engine, or fire right engine. The environment rewards the agent based on how efficiently it uses its resources to successfully land between two goal posts (without crashing). The problem is considered "solved" if the agent consistently achieves an average reward of 200 over 100 consecutive trials. If the lander crashes it receives a reward of -100 and the trial ends. We use a model written in Keras [15] and Theano [12] provided by the authors of [8]. Different than supervised-learning that uses validation accuracy as its performance metric, reinforcement-learning uses reward.

Hyperparameter Sets. To ensure fair comparison, we use the same set of hyperparameters for evaluation, i.e., using the same random search Hyperparameter Generator with the same initial random seed. The hyperparameter set consists of 100 configurations for both supervised and reinforcement learning experiments. Specifically, we explore up to 14 different hyperparameters for CIFAR-10 with the same hyperparameters and value ranges as in Table 3 of [20]. We explore 11 different hyperparameters for 'LunarLander" and we use ranges and values provided by the authors of the model [8].

Testbed. We conduct live GPU experiments for supervised-learning on a private 4-machine GPU cluster that we refer to as *private-cluster*. We co-locate the HyperDrive scheduler with one of the training machines in the cluster. Each machine is equipped with an Intel Xeon E5-2680 v2 2.80GHz CPU, 128 GB of memory, 10 Gbps network connectivity, and one Tesla K40m GPU. We use Ubuntu 14.04.5 LTS with Python 2.7.6, CUDA v8.0.44, and the CuDNN library v5.1.10. We use a version of Caffe 1.0.0-rc3 that we modified to report application metrics (e.g., accuracy) to a local Node Agent running on the same machine.

We conduct reinforcement-learning experiments on AWS. We use 15 c4.xlarge instances for training and a single m4.xlarge instance for running the HyperDrive scheduler. Each training machine uses Ubuntu 16.04.02 LTS, Python 2.7.12, Theano 0.9.0, and Keras 1.1.0. For suspending and resuming a configuration, we incorporate CRIU 2.6 into HyperDrive.

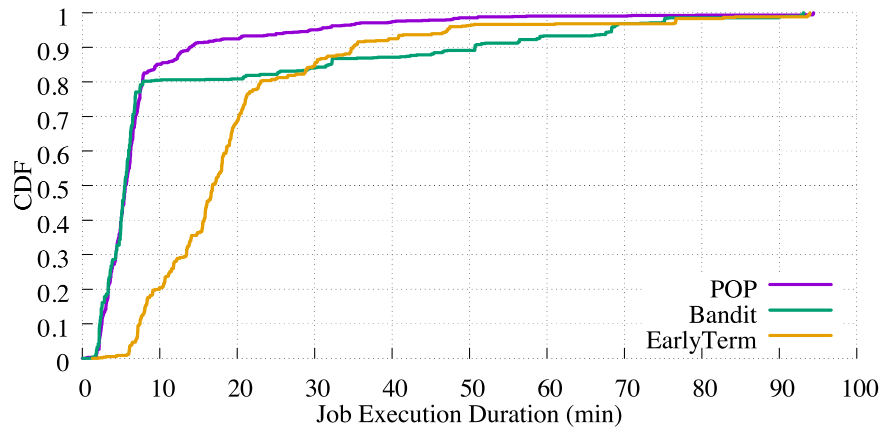

Figure 8.1: Job execution duration distribution comparing different scheduling policies with supervised-learning workload.

Non-Determinism. A challenge with evaluating scheduling algorithms for hyperparameter exploration is non-determinism that comes from the asynchronous nature of model training algorithms [47]. We observe that this non-determinism could vary model performance at a given epoch by up to 2%. To reduce the effect of this non-determinism, we run each experiment multiple times: 10 times for supervised learning and 5 times for reinforcement learning.

8.2 Supervised-Learning

8.2.1 Job Execution Duration

Figure 8.1 shows the distribution of job execution durations for POP, Bandit, and EarlyTerm. POP spends considerably less time across all jobs than the other policies, this is especially the case when looking at longer running jobs. Particularly we see that Bandit and EarlyTerm spend around 30 min or more on almost 15% of jobs, where POP spends 30 min or more on only 5% of jobs. We see in the following section that by spending less time overall executing less-promising jobs we are able to achieve improved performance.

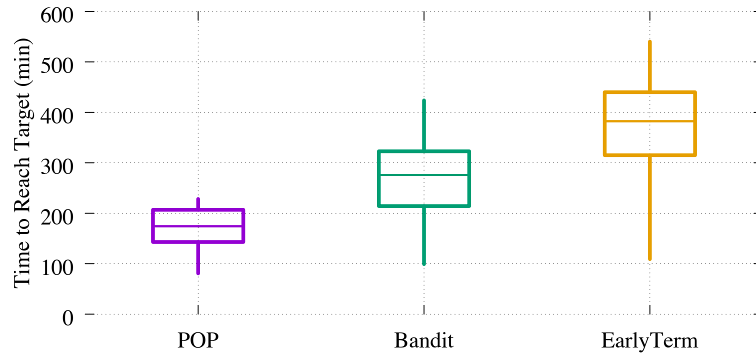

Figure 8.2: Time to reach target validation accuracy (CIFAR-10).

8.2.2 Scheduling Performance Comparison

We evaluate the performance of different policies by comparing the training time to reach a given target accuracy using the same cluster. We select a target accuracy of 77% based on the domain knowledge of our CIFAR-10 model [39], which is close to the best accuracy reported for this model. Our choice of this model is discussed in section 8.1. For each policy, we repeat the experiment 10 times. The results are presented in Figure 8.2 as box plots showing the different quartiles for achieving the target accuracy under each policy. On average POP reached the target accuracy in only 2.8 hours, whereas Bandit took 4.5 hours and EarlyTerm took 6.1 hours. POP outperforms Bandit by 1.6x and outperforms EarlyTerm by 2.1x. In addition, the difference between the minimum and maximum training times using POP is much smaller (around 2x) than Bandit and EarlyTerm. Even the worst performing run of POP is faster than the best case of the Bandit and EarlyTerm. This indicates that POP is not only faster in reaching target accuracy, but also offers more stable performance, thanks to its judicious classification and scheduling. We experimented with different training accuracy targets and different variations of CIFAR-10 and the observations are consistent. In the interest of space, we omit the results here.

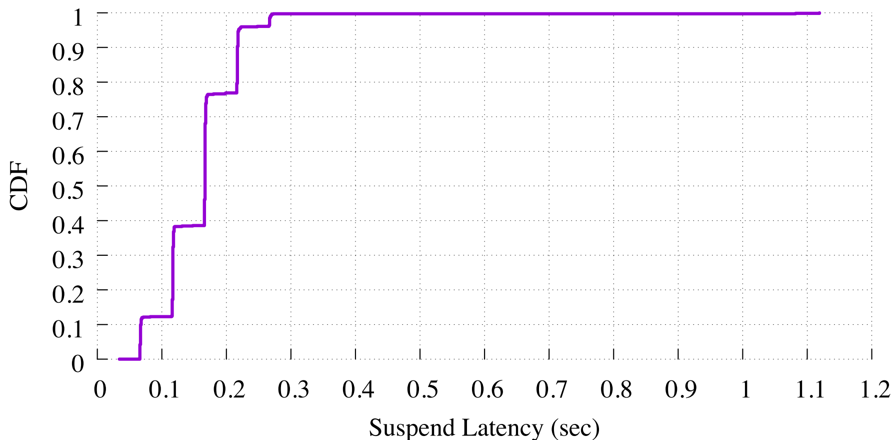

Figure 8.3: Suspend latency for supervised-learning workload

8.2.3 Scheduling Overhead

The advantages of HyperDrive with POP are at the cost of extra scheduling overhead compared to other approaches. The cost of suspending & resuming training jobs can incur higher overhead than other scheduling algorithms. Suspending training jobs involves capturing model state that enables later resumption of training. The captured model state of different jobs are sent to HyperDrive for storage and dissemination. Therefore, the overhead includes suspend/resume time and storage costs for model state. In this study we measure *suspend latency*, which measures the time between when the scheduler sends a suspend request to the Node Agent until the scheduler finishes storing the model state. We detail the snapshot latency and size behavior in Figure 8.3 and Figure 8.4. On average this latency is only 157.69 ms with a standard deviation of 72 ms. We observe the 95th percentile latency to be 219 ms and a maximum of 1.12 sec. In terms of the model state size we observe an average total size of 357.67 KB with a standard deviation of 122.46 KB. We observe the 95th percentile to be 685.26 KB and a maximum of 686.06 KB. We find in practice and show in our end-to-end evaluation results that these overheads show negligible impact on scheduling and training performance.

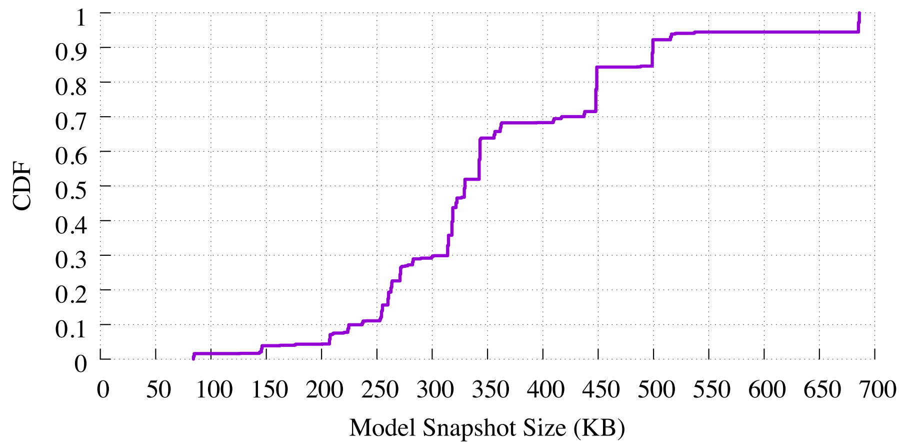

Figure 8.4: Model snapshot filesize for supervised-learning workload.

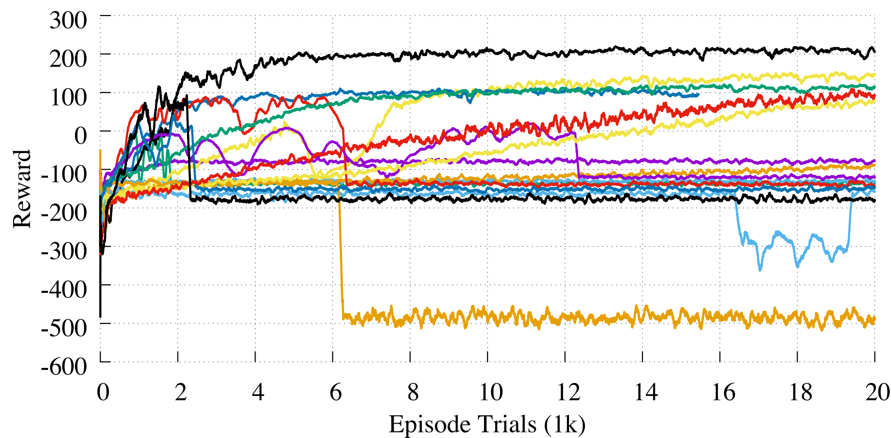

Figure 8.5: Performance of 15 randomly selected LunarLander model configurations over 20,000 episode trials.

8.3 Reinforcement-Learning

Figure 8.5 shows the performance of 15 randomly selected LunarLander configurations. Unlike supervised-learning, we observe that many jobs learn for some period of time and then experience what we call a "learning-crash", in which the reward falls and remains at or below a non-learning value. In the plot, the non-learning value is -100, which is related to the negative reward given by the environment when the lander crashes. We observe that over 50% of jobs are non-learning and should not be fully executed.

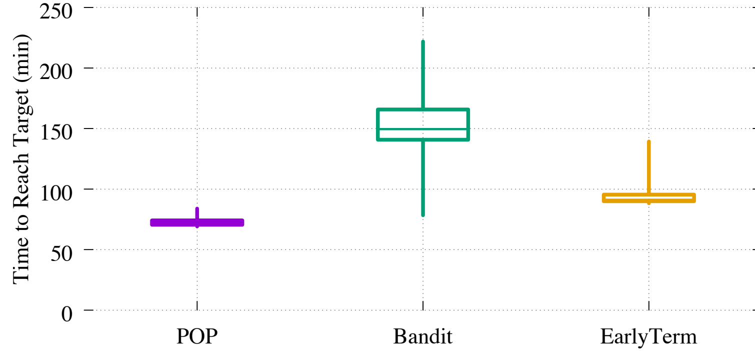

Figure 8.6: Time to reach target reward (LunarLander).

Reward values in the LunarLander task generally range between -500 and 300, in order for any scheduling policy to compare relative performance between configurations, we normalize all reward values using min-max scaling. We transform every reward value r as follows:

$$r_{norm} = \frac{r - r_{min}}{r_{max} - r_{min}}, \tag{8.1}$$

In our experiments, we use $r_{min} = -500$ and $r_{max} = 300$. The upper-bound range (r_{max}) is determined by the environment and task while the lower bound range (r_{min}) is determined empirically (we use this method) by observing a small number of poor performing runs or can by calculated the time allowed per episode and the maximum number of actions allowed.

8.3.1 Scheduling Performance Comparison

A priori target performance is common in many reinforcement-learning tasks. In LunarLander, the environment explicitly sets a "solved" condition that can be used as our target, i.e., an average reward of 200 over 100 consecutive trials.

Figure 8.6 presents the time to reach target results for each policy. We repeat the same experiment five times for each policy. We observe POP achieves a median time to target 2.07x faster than Bandit and 1.26x faster than EarlyTerm. Again, training time variations are much lower for POP compared to Bandit (9.7x smaller) and EarlyTerm (3.5x smaller) policies. These results show that

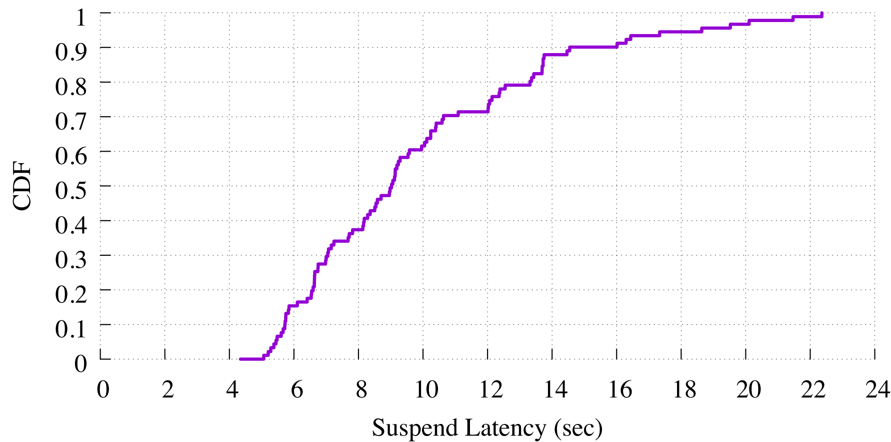

Figure 8.7: Suspend latency distribution for LunarLander workload.

compared to state-of-the-art approaches, HyperDrive with POP is faster in reaching target accuracy, and also more stable performance-wise for reinforcement learning.

8.3.2 Scheduling Overhead

We use CRIU to assist suspending/resuming training jobs. When a training job is suspended, all of its processes state is snapshotted and transferred back to HyperDrive. Instead of adding save/resume support to our model we use a more general approach using CRIU to snapshot the entire process state. We recognize that this method may incur higher overhead than a custom solution for our model. This study measures the overhead of suspending our LunarLander training job from the perspective of HyperDrive. Figure 8.7 and Figure 8.8 present the distributions of both suspend latency and model snapshot size. We see that model size does not exceed 43.75 MB and latency does not exceed a maximum of 22.36 sec, which is considerably small compared with job training time.

In summary, POP is faster in reaching target performance and more stable for different learning domains compared to state-of-the-art approaches such as Bandit and EarlyTerm.

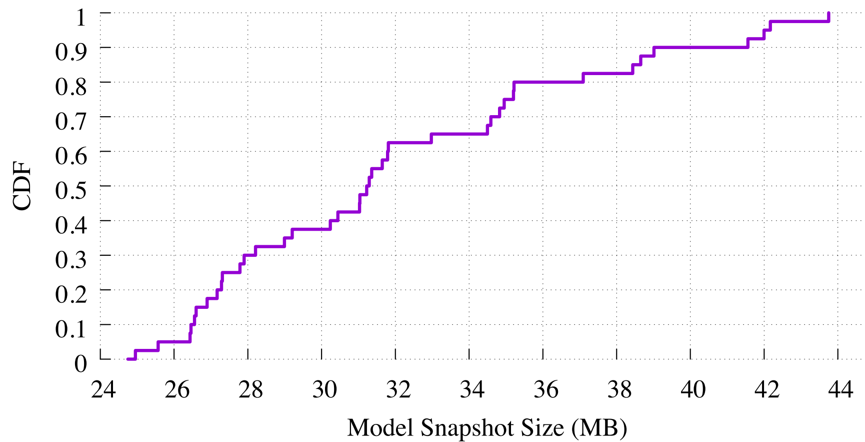

Figure 8.8: Suspend snapshot size distribution for LunarLander workload.

8.4 HyperDrive Sensitivity Analysis

In this chapter, we perform sensitivity analysis relating to the resource capacity and configuration order for both supervised and reinforcement learning using different polices. Due resource constraints, we opt for developing a simulator to perform sensitivity analysis. To ensure accurate simulation, we feed the simulator with traces collected from live system experiments.

8.4.1 Simulator

Our goal is to compare the scheduling efficiency (i.e., time to reach target accuracy) between different policies under different resource capacities and configuration orders. We develop a trace-driven simulator consisting of the following three main components, see Figure 8.9:

- **Trace Generator** collects the traces from live system experiments and creates a replayable workload that contains iteration timing and performance metrics. In addition, the Trace Generator can create traces by changing the configuration orders. This feature is useful to conduct sensitivity analysis of configuration orders.
- **Simulator Engine** is a trace-driven discrete event simulator that accurately emulates the execution process of HyperDrive, i.e., the order of configurations and the resource management logic.

- **Pluggable Scheduling Policy** dictates the scheduling decisions on configuration ordering and the resources allocated to different configurations over time.

To validate the accuracy of our simulator, we compare the simulation results with the live system results using different policies in Figure 8.10, which shows the time to reach target accuracy for LunarLander using 15 machines. We see the simulation results are quite accurate, i.e., compared to the live system results, the max error of simulation is only 13%, which is well below the error bar of live system results.

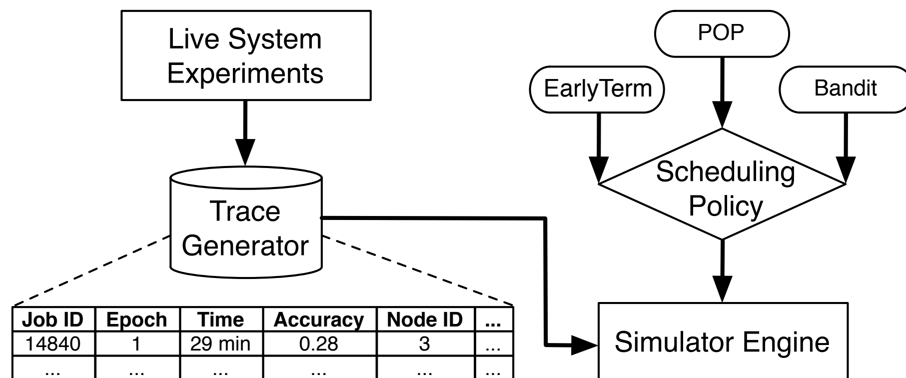

Figure 8.9: Simulator Design.

8.4.2 Supervised-Learning

All simulator traces for supervised-learning are collected from live AWS runs using 4 g2.2xlarge instances (one K520 GPU each)[1] for training and one m4.xlarge instance for the HyperDrive scheduler.

Sensitivity Analysis of Resource Capacity

We study the sensitivity of resource capacity (i.e., total number of machines) by comparing the time it takes to reach our target validation accuracy (77%) for different policies using CIFAR-10. Figure 8.11 presents the results of simulation experiments. As we would expect, the time to achieve

[1]The K520 GPU is slower than the K40m GPU used in our private-cluster.

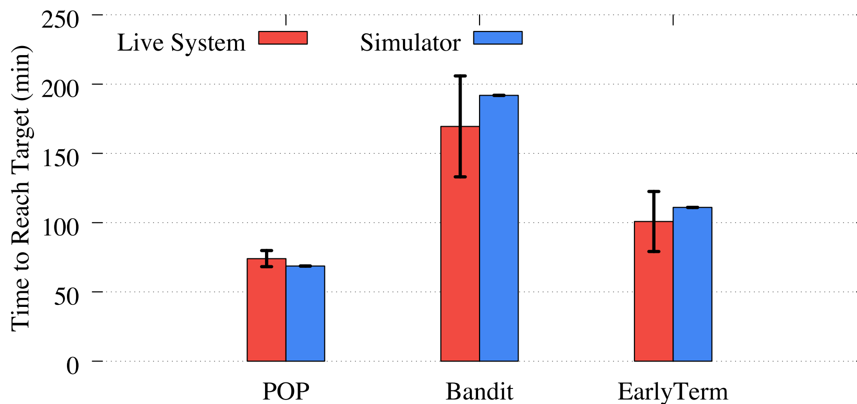

Figure 8.10: Simulator validation (LunarLander).

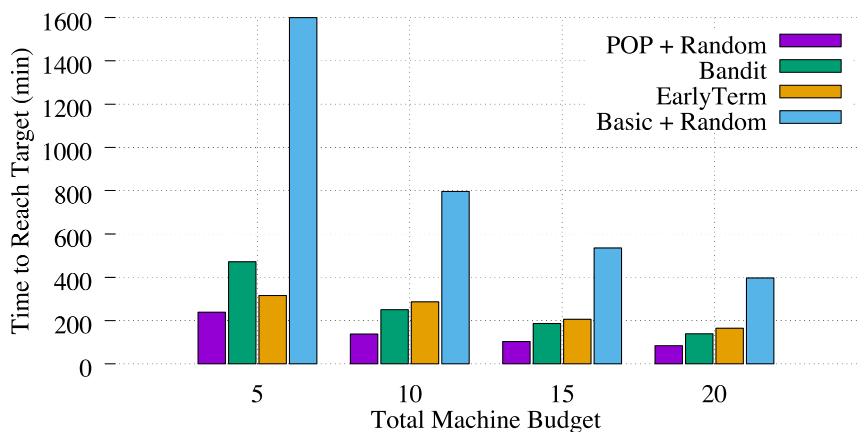

Figure 8.11: Resource capacities (CIFAR-10).

our target improves with more machines across all of our scheduling policies. POP always outperforms other policies under different resource capacities. In addition, with larger resource capacities, POP shows even more performance improvement to the second best policy. These results verify the effectiveness of our POP policy when using different amounts of resources.

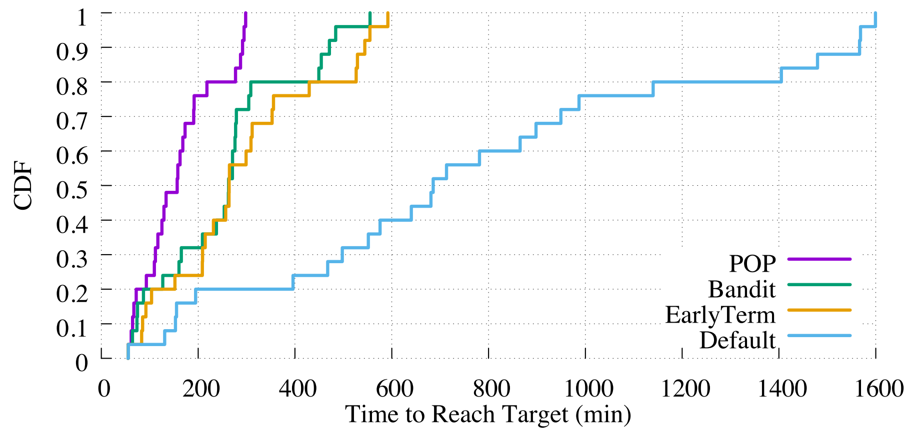

Figure 8.12: Random job orderings (CIFAR-10).

Sensitivity Analysis of Configuration Order

The configuration order that a scheduling policy sees can heavily impact experiment performance. For example, since configurations are generated randomly it's possible an exhaustive search technique could "luckily" pick the optimal configuration as the first configuration (or in the first batch of configurations) to explore, its performance can be as good as or even better than any sophisticated policies. In order to understand the sensitivity of configuration order for different policies, we run simulation experiments with 25 random configuration orders on 5 machines. The results presented in Figure 8.12 demonstrate the distribution of time to reach target validation accuracy under different polices. It is clear that POP yields much better performance in all percentiles. In addition, it is also more consistent in performance than other policies, e.g., POP has a maximum difference in completion time of 4.05 hours compared to Bandit with 8.33 hours, EarlyTerm with 8.50 hours, and Default with a staggering difference of 25.74 hours. These results suggest POP is less sensitive to configuration order and therefore is more reliable.

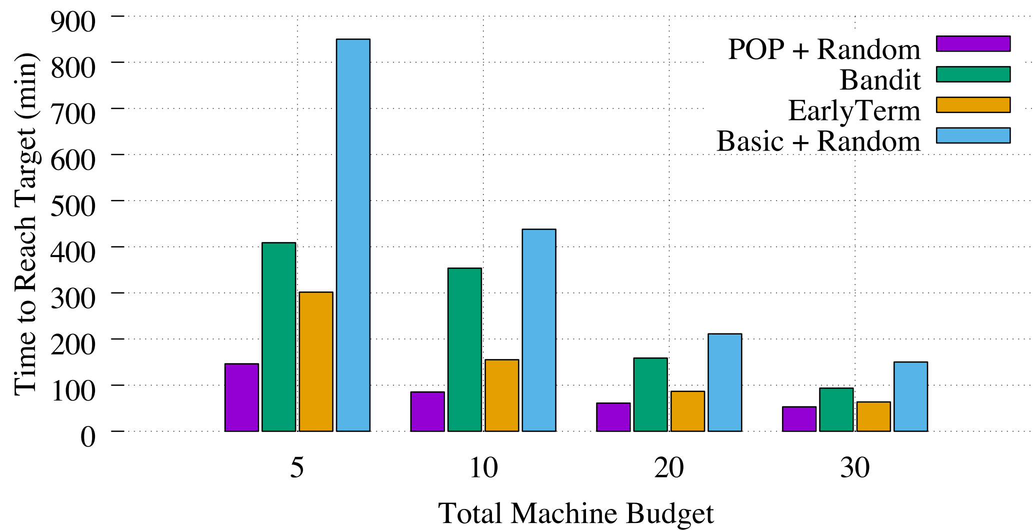

Figure 8.13: Time to reach target reward with LunarLander using different scheduling policies and resource capacities.

8.4.3 Reinforcement-Learning

We collected traces for reinforcement-learning experiments from running live experiments on AWS using 15 c4.xlarge instances for training and a single m4.xlarge instance for the HyperDrive scheduler. Each training machine uses Ubuntu 16.04.02 LTS, Python 2.7.12, Theano 0.9.0, and Keras 1.1.0. For suspend and resume support each training machines uses criu 2.6.

Sensitivity Analysis of Resource Capacity

We study resource capacity sensitivity in a similar way as in Section 8.4.2. We compare the time it takes to reach an average reward of 200 over 100 consecutive trials. Figure 8.13 presents the results of simulation experiments with a varying number of available machines. As expected, the time to achieve our target reward goal improves when additional machines are available for the experiment. We can see that POP consistently out performs other policies, regardless of the amount of resources available.

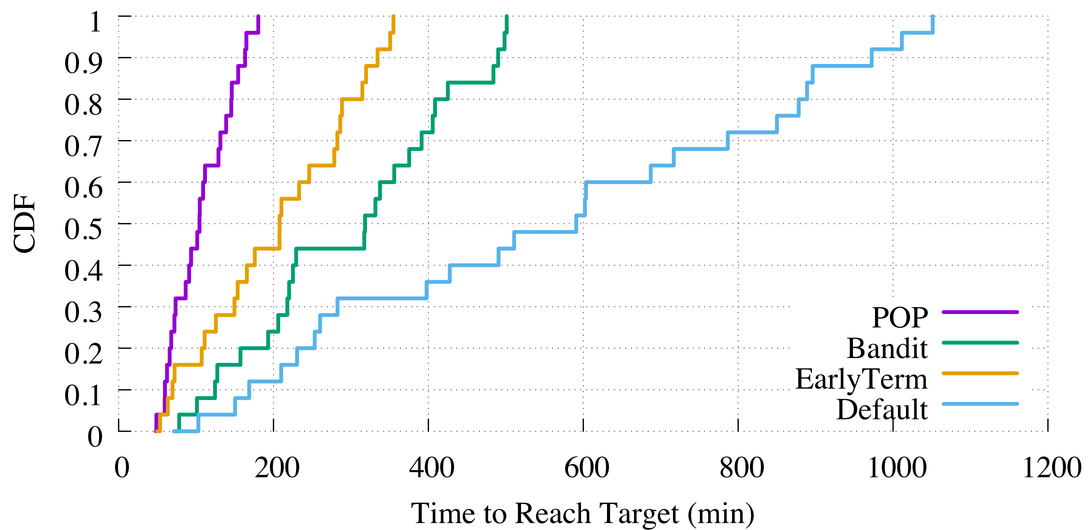

Figure 8.14: Time to reach target reward for LunarLander with 25 random configuration orders using different policies.

Sensitivity Analysis of Configuration Order

We examine the sensitivity of configuration order for our reinforcement-learning workload in a similar way as in Section 8.4.2. We run simulation experiments with 25 random configuration orders on 5 machines. The results presented in Figure 8.14 demonstrate the distribution of time to reach target reward with different polices. As with our supervised-learning workload we see that POP yields significantly better performance than under other policies, across all percentiles.

Chapter 9

Discussion of HyperDrive Considerations

9.1 Discussion

Epoch durations. Our POP policy assumes epoch durations remain relatively constant during training (see section 7.1). Epoch durations may differ between unique sets of hyperparameter values but for a specific configuration this duration remains relatively constant. This behavior is common in learning domains evaluated in this work (supervised and reinforcement), however we leave evaluating other domains that may experience non-constant epoch durations (e.g., genetic algorithms) to future work.

Learning curve prediction. Our POP policy relies on a learning curve prediction model. In this paper, we choose the model proposed in prior work [20] (which has been studied with extensive evaluation) for this component, we have found it to work well for the workloads we are using. In addition, we design the learning curve prediction module as a pluggable component of HyperDrive, so users can easily switch to other prediction methods as preferred. Learning curve prediction is an active area of research [20, 36, 54] and we foresee no issue using different approaches as research advances in this area.

User inputs. Our POP policy aims to achieve a training goal within specific time/resource

constraints, therefore it requires as input a maximum experiment time (T_{max}) and target performance (y_{target}). Specifying T_{max} requires some knowledge related to typical configuration training time, since a too small T_{max} may result in insufficient time to finish model training. Setting y_{target} is natural for domains with known goals, such as our LunarLander task (see section 8.3). However, if a y_{target} is unknown we have successfully used a dynamic target approach to automatically adjust y_{target} by gradually increasing the target once it is reached. In the interest of space, we leave the details and evaluation of this approach to future work. In our work with practitioners we have found that before starting hyperparameter exploration for their model they have a good idea about both of POP's required inputs.

POP, Bandit, and EarlyTerm policies all require a user-defined evaluation boundary (b) to be specified. Setting this value is a common problem in the space of early-termination policies. Like in prior work, b is model/domain specific and should reflect the time it takes to compute model performance and how long a user is willing to let a configuration execute before possible termination. We have found success with a heuristic of setting b to be between 5-10% of the max epoch for a job. We leave automatically setting this parameter to future work.

Ongoing Work. HyperDrive enables model owners to schedule resources based on monitored application-level metrics. Typically there is a primary metric being optimized (e.g., accuracy) which is what POP utilizes. However, additional metrics of concern can be impacted by hyperparameter choices as well, such as inference/serving latencies, model sparsity/compressibility, etc.

We have seen promising early results exploring hyperparameters specific to models that use Long Short-Term Memory (LSTM) units. We are working together with authors of recent work on improving CNN model sparsity [61]. The work aims to reduce the size of LSTMs structurally, for both storage saving and computation time saving, without perplexity loss (the primary metric for our task). This is done through the use of group Lasso regularization [66], which adds enforcement on each group of a model's weights. The method uses a new hyperparameter (i.e., λ) which makes a trade-off between sparsity and model perplexity. We have evaluated several state-of-the-art models from recent work [50, 69] with a new HyperDrive policy, exploring λ values (plus other hyperparameters) while monitoring both perplexity and a sparsity-related metric. We have seen

significantly reduced training times by enabling user-defined global termination criteria through HyperDrive's SAP API.

Lastly, we are working with Microsoft engineers to productize HyperDrive internally, which will continue improving HyperDrive's usability, scalability, and effectiveness. Hyperparameter exploration will continue to be an active research area but currently there are limited options to develop/evaluate parallel approaches that incorporate techniques along *both* hyperparameter generation and scheduling (e.g., early termination, suspend/resume).

9.2 Related Work

We discuss related hyperparameter optimization work for ML/DL models.

Learning Curve Prediction. Several pieces of related work present learning curve prediction models [20, 36, 54]. This prior work could be used as a drop-in replacement for the learning curve prediction model in our POP algorithm, we currently employ [20]. This line of work is complementary to POP, where we focus on how to effectively use them for classification and resource management.

Hyperparameter Generation. Several pieces of prior work [10, 33, 38, 52, 56] have moved beyond grid/random-based hyperparameter generation towards exploiting Bayesian optimization. Based on the accuracy of the configurations that have executed, they assign higher probability to exploit the areas of the hyperparameter space that contain likely-good or unknown configurations. This line of work is complementary to POP, where we can plug in different hyperparameter generation techniques. Our focus is to decide how much resources to map to the current set of configurations.

Sequential Search Algorithms. HyperBand [41] uses a multi-armed bandit approach for hyperparameter exploration. They use random search hyperparameter generation along with an aggressive bandit-based pruning technique. Swersky et al. [54] propose to suspend and resume uncompleted training jobs during model search, incorporating partial training information with learning curve

prediction. Their prediction model does not seem to work well for deep neural networks as discussed in [20]. In comparison, both prior work focus on using a sequential execution of models while POP is designed for speeding up model exploration using multiple machines and exploiting resource usage along the spatial dimension.

Parallel Search System. Recently, distributed computing frameworks such as YARN [55] and Spark [62] have recognized the importance of parallel hyperparameter exploration [32]. The proposed approaches enable parallel job execution for grid and random search. They do not support active monitoring of job metrics, early termination and dynamic resource allocation among configurations.

TuPAQ [53] is closely related, we present an overview of its bandit algorithm in section 7.6. POP employs job execution history for improved prediction and uses the predicted accuracy and confidence to enhance scheduling decision, while TuPAQ looks at the instantaneous accuracy of jobs when deciding who to terminate. In terms of system framework, TuPAQ focuses on a single learning domain (supervised) and computation framework (MLBase), while HyperDrive is designed as a comprehensive framework supporting different algorithms across frameworks and domains. For performance, our evaluation results (in chapter 8 and section 8.4) show POP consistently outperforms our Bandit policy, which is based on TuPAQ.

Chapter 10

Conclusion

This dissertation has presented the idea of infusing application-level information into the area of cluster resource management. In the pursuit of this idea we have shown our techniques benefit two high-level domains: big-data analytics and machine-learning model optimization systems.

10.1 Big-Data Analytics Systems

We first apply the concept of application-aware cluster scheduling in the context of big-data analytics systems. We observe several inefficiencies in existing cluster management framework designs in the presence of heterogeneous workloads that lead to poor utilization and degraded user-experience. To address these inefficiencies we employ worker queues to centralized frameworks to improve cluster utilization comparable to distributed frameworks. We then develop policies for active queue management, carefully choosing which task to execute next whenever a running task completes, with the goal of improved job completion times. The policies we develop are equally applicable to both centralized and distributed scheduling frameworks. We built a prototype of our techniques, Yaq, and deployed it on a large cluster. We experimentally evaluated Yaq using production and synthetic workloads. Yaq improves job completion time across all percentiles and, in particular, median job completion time by up to 9.3x, when compared to existing scheduling schemes, such as YARN, Mercury and an implementation of Sparrows batch sampling on Mercury.

10.2 Machine-Learning Model Optimization Systems

The second domain we apply our techniques to is the area of machine-learning model optimization systems. This work improves the efficiency of developing machine learning models by optimizing the problem of hyperparameter exploration. Our approach includes two techniques: (i) the POP scheduling algorithm and (ii) the HyperDrive framework. POP employs dynamic classification of model configurations and prioritized resource scheduling based on application-level metrics to discover high-quality models faster than state-of-the-art approaches. HyperDrive is a flexible framework that enables convenient evaluation of different hyperparameter exploration algorithms to improve the productivity of practitioners. We present experimental results that demonstrate the performance benefits of using POP and HyperDrive to develop high-quality models in both supervised and reinforcement learning domains.

10.3 Future Work

In this section we list a few areas of future work related to application-aware resource management. Specifically, we discuss the area of network and application load balancers and an extension area of our work in machine-learning optimization systems.

10.3.1 Application-Aware Load Balancing

Services like Envoy [42] and Maglev [21] are used to efficiently route and control traffic in micro-service oriented clusters. Similar to modern cluster schedulers like YARN, these services by design, attempt to abstract away as much application-level information as possible in order to isolate complexity. One of the many features that these services provide is the ability to rate-limit and load balance traffic across application services. A largely unexplored area on research in this area is how to set application-level priorities on traffic passing through these load balancers. Typically priority levels in load balancers are used to route traffic away from "unhealthy" application services [9]. However, we believe that we should be able to infuse application-level priorities into these load

balancers to improve end-to-end application-level quality of experience/service metrics. Similar to our work with Yaq and HyperDrive we should be able to identify and prioritize traffic that is related to high-level application-level goals such as meeting strict service-level agreements.

10.3.2 Multi-Metric Machine-Learning Optimization Systems

One largely unexplored area in this dissertation is the optimization of cluster resources for *multiple utility metrics* at once. In Yaq [46] we optimized cluster resources based on several application-level metrics such as task durations, remaining job time, etc. Whereas in HyperDrive [45] we optimized cluster resources based on a user-provided application utility metric (e.g., accuracy, reward). In the context of many machine-learning applications, once a data scientist has successfully trained a model to solve a task they will productize the model for answering future inference queries. Model inference in production environments require that a model can be evaluated in very strict time limits. These time limits are commonly referred to as service-level objectives (SLOs). These SLOs are based on high-level business objectives for large services such as web-search, or smaller environments such as mobile applications.

As shown in our HyperDrive [45] work, hyperparameter selection has a significant impact on the ultimate usability of a fully trained model. Properly tuned hyperparameter values can yield a model that performs significantly better in terms of accuracy, reward, etc. However, it turns out that certain hyperparameter values can positively or negatively impact the inference time of a trained model. This means that a model that performs very well on a user's primary utility metric may not meet the business objective (SLO) to be released to production. A potentially fruitful area of future work in this space would explore how to aggressively apply early termination strategies to model candidates in order to save time training models that will not meet a pre-defined SLO.

Bibliography

[1] A high performance, open-source universal RPC framework. `https://grpc.io`, 2017.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM*, 2013.

[4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *OSDI*, 2010.

[5] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice*. Recursive Books, second edition, 2014.

[6] `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`.

[7] `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`.

[8] Kavosh Asadi and Jason D. Williams. Sample-efficient Deep Reinforcement Learning for Dialog Control. *CoRR*, abs/1612.06000, 2016.

[9] Envoy authors. Envoy Priority Levels. `https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/load_balancing#priority-levels`, 2018.

[10] The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. `http://github.com/SheffieldML/GPyOpt`, 2016.

[11] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages I–115–I–123. JMLR.org, 2013.

[12] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU Math Compiler in Python . In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 3 – 10, 2010.

[13] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, October 2014. USENIX Association.

[14] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[15] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[16] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: A Modular Machine Learning Software Library. Idiap-RR Idiap-RR-46-2002, IDIAP, 0 2002.

[17] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[18] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *USENIX ATC*, 2015.

[19] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *SoCC*, 2015.

[20] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3460–3468. AAAI Press, 2015.

[21] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, 2016.

[22] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. Action Elimination and Stopping Conditions for the Multi-Armed Bandit and Reinforcement Learning Problems. *Journal of machine learning research*, 7(Jun):1079–1105, 2006.

[23] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[24] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*, 2011.

[25] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *USENIX ATC*, 2015.

[26] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 455–466, New York, NY, USA, 2014. ACM.

[27] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*, 2015.

[28] Apache Hadoop Project. `http://hadoop.apache.org/`, 2016. Online, accessed November 2016.

[29] `http://hadoop.apache.org/docs/r1.2.1/gridmix.html`.

[30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[31] Chi-Yao Hong, Matthew Caesar, and Brighten Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, 2012.

[32] Tim Hunter. Deep Learning with Apache Spark and TensorFlow. `https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html`, January 2016.

[33] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proc. of LION-5*, pages 507–523, 2011.

[34] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[35] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, Santa Clara, CA, July 2015. USENIX Association.

[36] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with Bayesian neural networks. *Proc. of ICLR*, 17, 2017.

[37] Oleg Klimov. Lunarlander-v2. `https://gym.openai.com/envs/LunarLander-v2`, 2017.

[38] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, 2014.

[39] Alex Krizhevsky. cuda-convnet. `https://code.google.com/p/cuda-convnet/`, 2017.

[40] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009. Technical report, University of Toronto.

[41] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: Bandit-based Configuration Evaluation for Hyperparameter Optimization. *Proc. of ICLR*, 17, 2017.

[42] Lyft. Envoy. `https://www.envoyproxy.io/`, 2018.

[43] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The Case for Tiny Tasks in Compute Clusters. In

Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems, HotOS'13, pages 14–14, Berkeley, CA, USA, 2013. USENIX Association.

[44] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[45] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 1–13, New York, NY, USA, 2017. ACM.

[46] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient Queue Management for Cluster Scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 36:1–36:15, New York, NY, USA, 2016. ACM.

[47] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[48] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1357–1369, New York, NY, USA, 2015. ACM.

[49] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[50] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional Attention Flow for Machine Comprehension. *arXiv CoRR*, abs/1611.01603, 2016.

[51] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 1996.

[52] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[53] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Automating Model Search for Large Scale Machine Learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 368–380, New York, NY, USA, 2015. ACM.

[54] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-Thaw Bayesian Optimization. *arXiv preprint arXiv:1406.3896*, 2014.

[55] Wangda Tan and Vinod Kumar Vavilapalli. Distributed TensorFlow Assembly on Apache Hadoop YARN. `https://hortonworks.com/blog/distributed-tensorflow-assembly-hadoop-yarn/`, March 2017.

[56] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 847–855, New York, NY, USA, 2013. ACM.

[57] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2), 2009.

[58] `http://tinyurl.com/fbcorona`.

[59] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[60] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM.

[61] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning Structured Sparsity in Deep Neural Networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2074–2082. Curran Associates, Inc., 2016.

[62] Lee Yang, Jun Shi, Bobbie Chern, and Andy Feng. Open Sourcing TensorFlowOnSpark: Distributed Deep Learning on Big-Data Clusters. `http://yahoohadoop.tumblr.com/post/157196317141/open-sourcing-tensorflowonspark-distributed-deep`, February 2017.

[63] Extend YARN to support distributed scheduling. `https://issues.apache.org/jira/browse/YARN-2877`.

[64] Queuing of container requests in the NM. `https://issues.apache.org/jira/browse/YARN-2883`.

[65] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari

Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. An Introduction to Computational Networks and the Computational Network Toolkit. Technical report, October 2014.

[66] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[67] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[68] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[69] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent Neural Network Regularization. *arXiv CoRR*, abs/1409.2329, 2014.

[70] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy H. Katz. DeTail: Reducing The Flow Completion Time Tail In Datacenter Networks. In *SIGCOMM*, 2012.