Good Things Come to Those Who Wait: Optimizing Job Waiting in the Cloud

Pradeep Ambati, Noman Bashir, David Irwin, and Prashant Shenoy
University of Massachusetts Amherst

ABSTRACT

Cloud-enabled schedulers execute jobs on either fixed resources or those acquired on demand from cloud platforms. Thus, these schedulers must define not only a scheduling policy, which selects which jobs run when fixed resources become available, but also a *waiting policy*, which selects which jobs wait for fixed resources when they are not available, rather than run on on-demand resources. As with scheduling policies, optimizing waiting policies requires *a priori* knowledge of job runtime. Unfortunately, prior work has shown that accurately predicting job runtime is challenging. In this paper, we show that optimizing job waiting in the cloud is possible without accurate job runtime predictions. To do so, we i) speculatively execute jobs on on-demand resources for a small time and cost to learn more about job runtime, and ii) develop a ML model to predict wait time from cluster state, which is more accurate and has less overhead than prior approaches that use job runtime predictions. We evaluate our approach on a year-long batch workload consisting of 14 million jobs, and show that it yields a cost and average wait time within 4% and 13%, respectively, of the optimal.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Cloud computing, job scheduling, cost-efficiency

ACM Reference Format:
Pradeep Ambati, Noman Bashir, David Irwin, and Prashant Shenoy. 2021. Good Things Come to Those Who Wait: Optimizing Job Waiting in the Cloud. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472883.3487007

1 INTRODUCTION

Batch job schedulers, such as Slurm [2] and LSF [11], execute a large fraction of the workload in high-performance computing (HPC) clusters and data centers. While these schedulers were originally designed to manage a fixed set of servers, they are now generally "cloud-enabled" and capable of autoscaling by programmatically acquiring virtual machines (VMs) from cloud platforms on demand to execute jobs [1]. Thus, these schedulers must not only schedule jobs on fixed resources, but also decide when to acquire and release on-demand cloud resources. Hybrid clouds often use cloud-enabled schedulers to "burst" into the cloud when their fixed private resources are fully utilized [21]. Cloud platforms have also begun to offer native cloud-enabled schedulers with autoscaling, such as Amazon Web Services (AWS) Batch [9] and Azure Batch [10].

Cloud-enabled scheduling differs from conventional scheduling on fixed resources in that cost, in addition to job waiting time, is a critical metric. Indeed, jobs submitted to a cloud-enabled scheduler never need to wait for fixed resources, since the scheduler can always immediately acquire on-demand resources to execute them. However, forcing some jobs to wait for fixed resources can lower cost by using fewer on-demand resources, which are generally more expensive than highly utilized fixed resources. For example, in AWS, reserved VMs are a form of fixed resource, and are ~40-60% cheaper than on-demand VMs when utilized fully over a 1-3 year term. In addition, once purchased, fixed resources represent a sunk cost that cannot be recovered.

As a result, cloud-enabled schedulers must not only define a scheduling policy, which selects which jobs run when fixed resources become available, but also a *waiting policy*, which selects which jobs wait for fixed resources, and for how long, when they are not available before running on on-demand resources. Waiting policies often mirror traditional scheduling policies, such as Shortest Job First (SJF). Prior work analytically models simple waiting policies, including All Jobs Wait (AJW), No Jobs Wait (NJW), Long Jobs Wait (LJW), and Short Waits Wait (SWW), and shows that combining LJW and SWW offers a much better cost-waiting time tradeoff than the others [13]. We provide background on waiting policies and their relationship to scheduling policies in §2.

Importantly, as with many scheduling policies, optimizing the waiting policies above requires *a priori* knowledge of job runtimes. In particular, LJW directly requires job runtimes, since it forces jobs with runtimes larger than some threshold to wait for fixed resources, but runs shorter jobs immediately by acquiring on-demand resources. LJW has a similar effect as SJF in prioritizing short jobs, but with the additional cost of renting on-demand resources. Likewise, SWW indirectly requires job runtimes, since it forces arriving jobs expected to wait less than some threshold for fixed resources to actually wait, but runs jobs expected to wait longer immediately by acquiring on-demand resources. In this case, a job's expected waiting time is dependent on the runtimes of the jobs ahead of it in the queue [15, 33]. Unfortunately, scheduling policies that require knowing job runtimes, such as SJF, are often not widely used because accurately predicting job runtimes remains challenging. Recent work highlights many reasons for the low prediction accuracy, including a lack of sufficient features for training machine learning (ML) models and non-stationarity in workloads that leads to inconsistent performance [25]. Directly implementing the waiting policies above suffers from the same challenges.

The primary contribution of this paper is showing that optimizing waiting policies for cloud-enabled schedulers is possible without accurate job runtime predictions, and can come close to the cost and waiting time achievable given perfect knowledge of job runtimes. Specifically, we show that it is possible to approach the optimal cost and waiting time of the waiting policies above when using an oracle with perfect knowledge of job runtime. To do so, we develop two techniques to optimize job waiting under LJW and SWW, respectively. Intuitively, optimizing these waiting policies in the cloud is simpler than optimizing scheduling policies for fixed resource because i) there is no hard resource constraint, and ii) our waiting policy predictions require only binary classification, i.e., where a job's running or waiting time crosses a threshold, which does not require absolute model accuracy.

- **Speculative Execution**. We first leverage the availability of on-demand cloud resources to speculatively execute *all* jobs for some time to learn more about each job's running time before deciding whether to run it on fixed or on-demand resources. *This technique informs LJW, and is effective because, in many batch workloads, most jobs are short, but the few long jobs account for most of the computation.* Thus, the additional cost of speculatively executing the few long jobs incorrectly before restarting them on fixed resources is small. This additional cost is also smaller than using LJW with highly accurate job runtime predictions, as low as 10% error.
- **ML-based Waiting Time Predictions**. We next develop a ML model for predicting job waiting time, which can inform SWW. While accurate waiting time predictions are not widely used by scheduling policies, since they are only informative for users and do not improve scheduling performance, they are critical to optimizing SWW. Unlike prior work that uses job runtime predictions to estimate waiting time, e.g., by simulating the schedule based on the runtime predictions [15, 33], our ML model uses cluster state as its input, e.g., cluster size, number of jobs running and waiting, how long jobs have already run and waited, etc. *This technique is effective for SWW, and is more accurate than using job runtime predictions, largely due to the law of large numbers, as cluster state incorporates the attributes of many jobs rather than individual jobs.* We show that this approach is effective at achieving near the cost and waiting time of SWW with perfect knowledge of waiting time.

Our hypothesis is that combining speculative execution and ML-based waiting time predictions can achieve cost and waiting times that are close to optimal LJW and SWW with perfect job running and waiting time predictions. In evaluating our hypothesis, we make the following contributions.

Waiting Policy-based Schedulers. We provide background on cloud-enabled schedulers, contrast them with conventional schedulers for fixed resources, and highlight the importance of waiting policies. We present two waiting policies, LJW and SWW, describe their benefit to cost and waiting time, and discuss their impracticality due to requiring *a priori* knowledge of job runtime.

Optimizing Job Waiting. We show how to optimize LJW and SWW without requiring job runtime predictions using speculative execution (for LJW), and ML-based waiting time predictions from cluster state (for SWW). Speculative execution is effective because its costs are low—re-starting some long jobs—and its benefits are high—decreasing the waiting time of short jobs and not incurring additional costs to run long jobs on on-demand resources. Our ML model is effective because a large cluster's state, i.e., its set of running and queued jobs, correlates well with job waiting time.

Implementation and Evaluation. We implement our waiting policies in a trace-driven simulator, and evaluate them on a year-long batch workload consisting of 14 million jobs run on a 14.3k-core cluster. Specifically, we show that using both LJW with speculative execution and SWW with ML-based wait time predictions yields a cost and waiting time within 4% and 13%, respectively, of using LJW and SWW with perfect knowledge of job runtime. Our evaluation in §5.3 also shows that our insights and results generalize to a recently released Google workload [36].

2 BACKGROUND

Below, we first provide an overview of waiting policies, then discuss their interaction with scheduling policies, and finally present the specific context that motivates our work.

2.1 Waiting Policy Overview

Our work focuses on batch schedulers that schedule jobs on a cluster of servers, with a particular emphasis on cloud-enabled batch schedulers that schedule jobs on a mix of fixed servers and dynamically acquired on-demand servers. We assume a cloud-enabled batch scheduler that acts as a customer of a cloud platform, and services jobs on behalf of a set of users. These users submit jobs to the cloud-enabled scheduler with specified resource requirements, such as their number of required cores and memory. The scheduler can either schedule these jobs to run on any of s fixed resources, or acquire on-demand resources to run the jobs for an additional price p (in dollars per unit time). If the fixed resources are fully utilized, then the jobs scheduled to run on them must wait in a queue for them to become available. Since fixed resources are a sunk cost, scheduling jobs on them is cheaper than using on-demand resources, but under heavy load can significantly increase the average and maximum job waiting time. Cloud-enabled schedulers can explicitly control the cost-waiting time tradeoff by defining a *waiting policy*, which determines which jobs wait for fixed resources, and for how long, before acquiring on-demand resources.

Recent work by Ambati et al. [13] analyzed the cost and waiting time behavior of various waiting policies using queuing theory. The work developed analytical models of multiple waiting policies by extending an $M/M/s$ queuing system to quantify their tradeoff between cost, waiting time, and fixed resource provisioning. As we discuss, the waiting policies that offer the best tradeoff require *a priori* knowledge of job running times and waiting times, which the authors assume are available with perfect accuracy. Given such perfect waiting policies, the authors focus primarily on optimizing the provisioning of fixed resources s to minimize cost for a given workload based on the price differential between fixed and on-demand resources. For example, in Amazon's Elastic Compute Cloud (EC2), a 3 year reserved VM utilized 100% of the time costs 60% less than renting an equivalent on-demand VM over the same period. Thus, purchasing a 3-year reserved VM that is utilized more than 40% of the time is cheaper than renting an equivalent on-demand VM.

In this paper, we instead assume that the fixed resources s are given, and focus on optimizing job waiting (and scheduling) in practice where *a priori* knowledge of job running and waiting times is generally not available. In much of our evaluation, we provision fixed resources s near the optimal, assuming a waiting policy with perfect knowledge, which is simple to empirically determine offline for a given workload and scheduling policy. Ambati et al. [13] also provide a solution for the optimal fixed resources s in the context of a $M/M/s$ queuing model for multiple waiting policies. Our primary goal is then to achieve an overall cost and average job waiting time near that of the optimal waiting policy that has perfect knowledge of job running and waiting times. We summarize below the two waiting policies from [13] that offer the best cost-waiting time tradeoff. Note that, unlike scheduling policies, these two waiting policies can be applied concurrently by running a job on on-demand resources if it does not satisfy the criteria for waiting in either policy.

Long Jobs Wait (LJW). LJW forces an arriving job to wait for fixed resources if its running time is greater than some threshold t. LJW is work-conserving, so arriving jobs always run on fixed resources if available. In LJW, short jobs also never wait—they run on fixed resources, if available, or on dynamically acquired on-demand resources, if not. LJW's intuition is that longer jobs should be able to wait longer for fixed resources than shorter jobs, since this waiting is a smaller fraction of their running time. LJW has a similar effect as SJF scheduling in prioritizing short jobs, and thus offers a good cost-waiting time tradeoff. Since, in many batch workloads, short jobs are a small fraction of the overall computation, the additional cost of running them on on-demand resources is not high, and, as in SJF, immediately running short jobs significantly improves average job waiting time.

Short Waits Wait (SWW). SWW forces an arriving job to wait for fixed resources if its waiting time would be less than some threshold time b; otherwise, it immediately provisions an on-demand resource to execute the job without waiting. SWW yields the same cost as an equivalent policy that requires all jobs to wait for fixed resources, but provisions on-demand resources to execute jobs once they have waited b time, since the same set of jobs run on on-demand resources in both cases [13]. However, SWW yields a significantly lower waiting time, especially under heavy load, since jobs that would have waited b time never wait, but run immediately. Thus, SWW is cost-neutral, but optimizing it can improve average job waiting time.

While we analyze LJW and SWW in isolation in §3, we intend them to be used in combination. There is no need to choose between these two waiting policies. The choice of parameters is subjective, and based on users' sensitivity to cost and waiting time, since waiting policies define a tradeoff between two. For example, if users have no cost sensitivity, then they need not wait, i.e., by setting $t=\infty$ and $b=0$. We quantify cost-waiting time tradeoff for our waiting policies and workloads in the next section.

2.2 Scheduling Policy Interaction

Waiting and scheduling are independent, but related, policies that a cloud-enabled batch scheduler can define. The waiting policy determines whether a job should wait for fixed resources or run on on-demand resources, while the scheduling policy determines which waiting job to execute next and which server it should execute on. Since waiting policies are defined independently of the scheduling policy for fixed resources, they can be used alongside any scheduling policy. In this paper, we focus specifically on the tradeoff between cost and job waiting time for a cloud-enabled job scheduler, where all jobs have the same priority level but may have different resource requirements and running times. This is the key tradeoff that waiting policies expose, and we focus on it in isolation to better understand it. However, in practice, cloud-enabled schedulers may also incorporate other criteria in their objective function, such as priority levels, resource type constraints, deadlines, and flexibility in their resource amount and degrees of parallelism. While these additional criteria may also affect the cost-waiting tradeoff, quantifying their effect is outside the scope of this paper.

The choice of scheduling policy — for a given workload, waiting policy, and fixed resources s — has little effect on cost, since cost is largely a function of fixed resource utilization, and any work-conserving scheduling policy will utilize the fixed resources at a similar level, albeit by selecting jobs to run in a different order. For jobs that run on fixed resources, the scheduling policy for a cloud-enabled scheduler behaves the same as it would for a conventional scheduler that only uses fixed resources. For example, SJF still minimizes average job waiting time among all scheduling policies for jobs that run on fixed resources. However, waiting policies, such as LJW, can mitigate some of the waiting time advantage SJF has over simpler scheduling policies, such as first-come-first-serve (FCFS), by also prioritizing short jobs at a small cost. Waiting policies like SWW also address SJF's primary drawback of starving long jobs, since under SWW jobs never wait longer than b. Thus, optimizing waiting policies can improve the performance of existing scheduling policies.

Of course, in deciding which jobs to schedule next on fixed resources, conventional schedulers also implicitly decide which jobs must wait. However, conventional schedulers for fixed resources *cannot control* how long these jobs wait for fixed resources without affecting running jobs, e.g., by preempting them. Cloud-enabled schedulers are different: they *can control* how long jobs wait for fixed resources by deciding when to dynamically acquire on-demand/spot resources to run them. Thus, our concept of a waiting policy does not exist for a conventional scheduler that only uses fixed resources. Beyond [13], we are aware of no prior work that examines waiting policies for cloud-enabled schedulers.

2.3 Motivating Context and Baselines

This paper's methodology is empirical, and focuses on optimizing the workload of a large production batch cluster that services roughly 14 million jobs per year. The cluster currently consists of ~14.3k cores, uses the LSF job scheduler, and is not cloud-enabled. Our motivation is to understand how to operate such a cluster on a cloud platform, and its impact on cost and job waiting time. We have job traces from the past few years that include each job's submission time, user ID, maximum running time limit, requested number of cores and memory, completion status (finished, terminated, or cancelled), and running time. The max running time limit

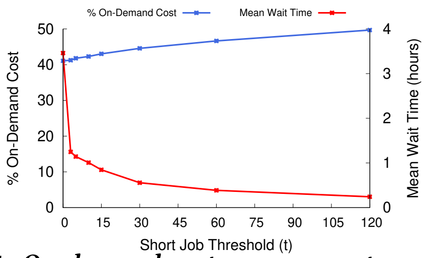


Figure 1: *On-demand cost, as a percentage of fixed resource cost, and average waiting time as a function of LJW's short job threshold t. As t increases, waiting time drops steeply, while cost increases modestly.*

is not indicative of a job's actual running time, and is typically many orders of magnitude larger. While the traces do not record job waiting time, we estimate the average waiting time would be ~8.1 hours using work-conserving FCFS scheduling, which schedules the first job near the front of the queue capable of running on the available resource. Resources are never idle if there is a job near the front of the queue that can run on them. The waiting time under non-preemptive, work-conserving SJF would be much lower at ~0.6 hours given the workload's large number of short jobs, but requires accurate predictions of job running time.

We consider executing the workload above with a cloud-enabled scheduler on EC2 using the SWW and LJW waiting policies combined with either non-preemptive, work-conserving FCFS or SJF scheduling. For fixed resources, we assume the use of 3-year reserved `m5.16xlarge` VMs, which each have 64 cores and 256GB memory. We choose larger VMs to mitigate the impact of imperfect packing of variable-sized jobs onto VMs. Since jobs request multiple cores, we quantify job length using total core-time rather than absolute running time. We focus on core-time, since cores are more constrained than memory. Our scheduler packs jobs onto fixed resources using a best-fit policy based on cores. When acquiring on-demand VMs to execute a job, our scheduler selects the smallest and cheapest VM within the `m5` family that satisfies the job's resource requirements. We also discuss using cheaper spot VMs instead of on-demand VMs.

For our baseline, we choose SWW's waiting time threshold b to be 24 hours, and LJW's runtime threshold t to be 15 minutes. Our choice for b is subjective: a higher b decreases cost, but increases waiting time, and there is no optimal value. We chose a 24-hour maximum waiting time because it seems reasonable that no job should wait longer than 1 day to run. Our choice for t=15m is based on Figure 1, which graphs the additional cost of using on-demand resources (on the left y-axis), as a percentage of the cost of fixed resources (discussed below), and average waiting time (on the right y-axis) as a function of t. The graph shows that LJW's cost is mostly flat, while the average waiting time initially decreases sharply and then flattens out. After the initial decrease, LJW's cost-waiting time tradeoff remains relatively constant.

Assuming the waiting policies above with perfect knowledge of job running and waiting times, we empirically determined that the optimal number of `m5.16xlarge` reserved VMs that minimizes cost for our workload was 150 for both SJF (with perfect knowledge of job running times) and FCFS scheduling. That is, adding another reserved VM, as a fixed resource, would not be utilized more than 40% of the time, and would not justify its cost. Thus, we set our baseline for fixed resources s=150. For context, a cloud-enabled scheduler using LJW and SWW parameterized above with s=150 would cost 5% less overall, when combining the amortized fixed resource and on-demand cost, than the current fixed size cluster, which is equivalent to using 225 `m5.16xlarge` VMs, and yield an average waiting time of 0.85 hours when using work-conserving, non-preemptive FCFS scheduling. This waiting time is a similar order of magnitude as SJF's waiting time of 0.6 hours on the current-size cluster; incurs a lower cost; and, as we show, is achievable without highly accurate predictions of job runtime. Finally, while we choose b=24h, t=15m, and s=150 as baselines, our general insights are applicable at any values of these parameters, and especially for smaller s, since, as with scheduling, optimizing job waiting become more important under constraint. Our evaluation varies b, t, and s from our baselines.

3 DESIGN

Directly implementing SJF, LJW, and SWW in practice is challenging because it requires predictions of job runtime. Our primary goal is to come as close to SWW and LJW waiting policies without *a priori* knowledge of job waiting or running times. To better understand the performance of these policies in practice, we first trained and evaluated multiple simple ML models, including linear regression, random forest, support vector regression (SVR), and a neural net, to predict each job's runtime from its characteristics known at submission, as represented in our LSF batch traces. Our models' input features included each job's submission time, user ID, maximum running time limit, and requested number of cores and memory, while the output was the job's running time. We trained the models on data from 10 million jobs over 9 months, and evaluated them on a separate timeframe of 4 million jobs over 3 months.

Figure 2(a) shows the results for each model, where the y-axis is the mean absolute percentage error (MAPE) in predicting a job's runtime at submission time. As expected, the error is quite high for all models largely because our batch traces record only a few features for each job, so the models have little data with which to distinguish jobs. Also, most jobs are short, so even relatively small absolute prediction errors result in large percentage errors. With few data features and mostly short jobs, the models predict the runtime of most jobs as short, and thus the error in predicting long jobs' runtime is especially high. Even if more distinguishing job data existed and was available, such as executable name, input parameters, etc., the set of users submitting jobs to the scheduler changes over time, which decreases the accuracy of models trained on different users and jobs. While continuously updating models online is possible, it is not always effective due to a lack of sufficient new data. These simple observations are not new, and have been observed in many other batch workloads. Recent work highlights these characteristics as being among the reasons ML-based job runtime prediction models tend to be highly inaccurate and are not widely used by cluster schedulers [25].

Of course, our LJW waiting policy only requires classifying jobs to be above or below some threshold t. Thus, Figure 2(b) evaluates these models' binary classification accuracy using the Matthews Correlation Coefficient (MCC), which is the best single measure of binary classification performance. The MCC's values are in the range −1.0 to 1.0, with 1.0 being perfect prediction, 0.0 being random prediction, and −1.0 indicating the prediction is always wrong. The results show that the models are not much better than random predictions, as the MCCs are all near 0.

To get a sense of how effective (or ineffective) such models are in practice, we used the best model above to simulate SJF on our current fixed-size cluster. Recall from §2.3 that the average job waiting time under SJF with perfect knowledge of job runtime is 0.6 hours on the current fixed-size cluster (equivalent to 225 `m5.16xlarge` VMs). However, simulating SJF using our ML model to predict job runtimes results in an average waiting time nearly 3× higher at 1.71 hours. For context, a random job next policy yields an average waiting time of 3.1 hours, so our job runtime prediction model yields an average waiting time for SJF roughly mid-way between using perfect predictions and random predictions. Interestingly, despite our ML model's poor prediction accuracy, it does appear to have better accuracy with respect to ordering jobs. Note that a random job next policy has a much lower waiting time than FCFS (at ∼8.1 hours) because it is more likely to select one of the large number of short jobs to run.

While the ML models above are simple, and may not be the most accurate, they illustrate long-standing issues with predicting job runtime in batch workloads. As we discuss, improving the accuracy of these models is *not necessary* to optimize job waiting for cloud-enabled schedulers. This is due, in part, to LJW and SWW's use of a threshold to make decisions. LJW classifies a job as long, if its running time exceeds a threshold t, and as short otherwise, while SWW similarly classifies a job's waiting time as short if it is less than b. In both cases, the decision depends on whether the

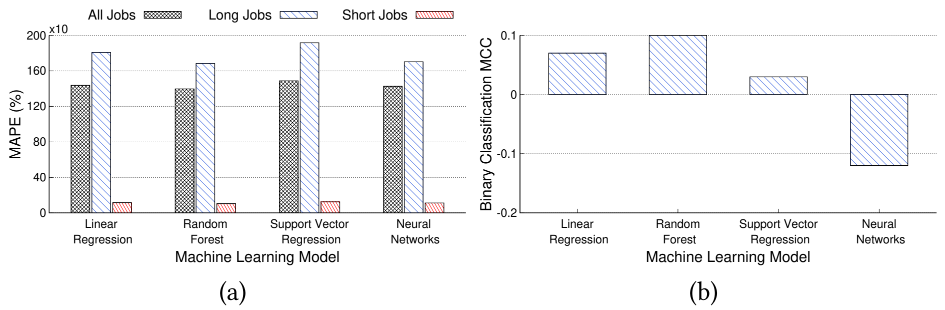


Figure 2: ***MAPE (a) and MCC (b) of multiple ML models for predicting job runtime from features in our batch trace.***

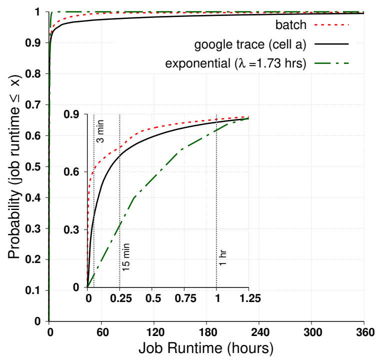


Figure 3: ***CDF of job runtime for our batch workload, an exponential distribution with the same mean, and a widely-used Google job trace [36]. The inset graph magnifies this CDF for job runtimes up to1.25 hrs.***

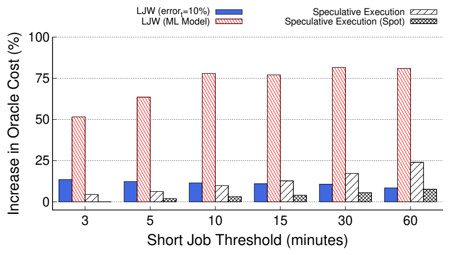


Figure 4: ***Percentage increase in cost compared to using LJW with a job runtime prediction oracle for multiple LJW variants (bars) and multiple short job thresholds.***

estimated value is above or below a threshold. Thus, even when job runtimes cannot be estimated (or even ordered) accurately, the waiting policy is correct as long as the job (in LJW) or wait time (in SWW) is estimated accurately with respect to its threshold.

3.1 Optimizing LJW: Speculative Execution

Cloud-enabled schedulers can always acquire on-demand resources at an additional cost to execute jobs. We leverage this capability to optimize the LJW waiting policy for a small additional cost. As with SJF, LJW requires *a priori* knowledge of job running time to make decisions about which jobs wait, and which jobs run, on on-demand resources. Instead of using a job runtime prediction model like those above to make this decision, our approach is to acquire on-demand resources to run *all* jobs immediately if no fixed resources are available when a job arrives. If a job's running time is less than t, then it will simply complete without incurring any additional cost or waiting time compared to if the scheduler had perfectly predicted its runtime. However, once a job runs on on-demand resources for t time, based on LJW, it is classified as a long job and should wait to run on fixed resources. In this case, the scheduler kills the job, releases the on-demand resources, and queues the job to run on fixed resources. We do not assume the scheduler can checkpoint, migrate, and restore jobs, since most schedulers do not support it.

The benefit of speculative execution for LJW is that it always handles short jobs correctly, by running them on on-demand resources to completion (when fixed resources are unavailable), but it incurs an additional cost of $t \times p$ for long jobs compared to if the scheduler had perfectly predicted job runtimes. Cloud-enabled schedulers can effectively leverage speculative execution to buy some information about job running time. The approach is cost-effective in practice if most of the jobs are short, but most of the resources are used by long jobs. This is case for many batch workloads, including ours. Figure 3 shows a cumulative distribution function (CDF) of job runtimes for our cluster's workload with dotted vertical lines at runtimes of 3 minutes, 15 minutes, and 1 hour. The lines show that 62% of jobs have a runtime of less than 3 minutes, 70% have a runtime less than 15 minutes, and as many as 95% have a runtime less than 1 hour. The graph also has lines for a widely-used publicly-available Google trace [36] and an exponential job runtime distribution with the same mean as our trace. The Google trace exhibits the *same* characteristics and general trend as our batch workload. While this skewed runtime distribution makes it challenging to identify the few long jobs, it also makes speculative execution cost-effective. Since both real-world workloads have a significantly higher fraction of short jobs than the exponential distribution, speculative execution in practice is likely to be much more effective than queuing models suggest [13].

To illustrate, Figure 4 plots the LJW threshold t on the x-axis, and the increase in on-demand VM cost, as a percentage

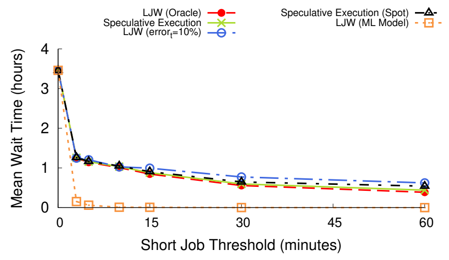


Figure 5: *Average waiting time for multiple LJW variants (lines) and multiple short job thresholds.*

of the same cost when using LJW with an oracle that has perfect knowledge of job runtime predictions. This graph uses non-preemptible, work-conserving FCFS scheduling. The graph compares the oracle with four different approaches: LJW using our best ML model from Figure 2, LJW using a hypothetical ML model with 10% average error in job runtime predictions, speculative execution on on-demand VMs, and speculative execution on spot VMs. The graph shows that LJW using our ML model performs the worst, resulting in a 50-75% increase in the cost of using on-demand VMs. In contrast, speculative execution on on-demand VMs incurs a much lower cost, especially for small thresholds. Speculative execution has a similar cost to using a hypothetical job runtime prediction model with 10% average error at our baseline of t=15 minutes (and below). As t increases, speculative execution's cost also increases, as there are fewer jobs at these longer durations.

We can further decrease the cost of speculative execution using spot VMs instead of on-demand VMs. Speculative execution is well-suited to using spot (or preemptible) VMs, which are offered by all of the major cloud platforms [3, 5, 6]. These VMs are ~70% less than on-demand VMs, but may be revoked by the cloud platform at any time. Recent work empirically models the revocation characteristics of preemptible and spot VMs [24], and shows that their mean time to revocation is usually on the order of hours or days, especially if the VM is not revoked soon after initialization. We use the same revocation model as [24] for Figure 4. With spot VMs, the longer a job runs, the more likely it will be revoked, and have to restart. However, since speculative execution only runs jobs for a short period, the probability of a spot revocation is low, making spot VMs effective at reducing cost for a small increase in risk. As the graph shows, speculative execution on spot VMs further lowers its cost.

Figure 5 similarly plots the corresponding average waiting time for each scenario. The graph shows that the average waiting time decreases as the LJW threshold t increases in all cases. LJW using our ML model has a near zero waiting time because it predicts nearly all jobs are short, and thus always runs them on on-demand VMs without waiting, which incurs a high cost. The other approaches yield an average waiting time close to, but slightly higher than, LJW using an oracle.

Feature	Description
cluster-cpu-util	Average CPU utilization of fixed resources
cluster-mem-util	Average memory utilization of fixed resources
runq-size	Number of running jobs on fixed resources
waitq-size	Number of jobs waiting for fixed resources
runq-mean-cpu	Mean CPU resource demand of jobs running on fixed resources
runq-mean-time	Mean time running (up to now) for jobs running on fixed resources
waitq-mean-cpu	Mean CPU resource demand of waiting jobs
waitq-mean-time	Mean time waiting (up to now) for jobs in queue
num-cores	Number of CPU cores requested by the new job

Table 1: *Cluster state features used for training our ML-based waiting time prediction models.*

Speculative execution on spot VMs yields a slightly higher average waiting time compared to using on-demand VMs due to the extra waiting time caused by spot revocations, which require re-starting jobs.

The results above indicate that, for batch workloads with job runtime distributions skewed towards short jobs, speculative execution mitigates the need to accurately predict job runtimes, and that such predictions would need to have less than 10% error to yield a similar cost and waiting time (for our baseline parameters). Achieving such low model error in practice is unlikely. We also examined using a speculative execution time t_s that is less than the LJW threshold t, and then making a job runtime prediction once a job has run for t_s time (and given that knowledge). However, given the knowledge that a job has run for $t_s \leq t$ time did not increase the accuracy of our runtime prediction models above, and did not approach the low ~10% error required to surpass speculative execution. Thus, we always set our speculative execution time equal to the LJW threshold t.

3.2 Optimizing SWW: Machine Learning

We next focus on optimizing SWW using an ML-based model for predicting a job's waiting time. There is less prior work on predicting waiting times for conventional schedulers, since these predictions typically serve only to inform users, but generally do not improve scheduling for fixed resources. However, prior work has explored predicting queue waiting times within the context of certain scenarios where it is more than just informative, such as when users can choose between multiple queues or clusters [15, 27], or when users can alter their requested resources *post facto* to shorten waiting time [34]. This prior work focuses on conventional scheduling for fixed resources, and not cloud-enabled scheduling.

A common approach for estimating queue waiting time is to use predictions of job running time to simulate the schedule forward [28, 33]. This approach, of course, is dependent on accurate job runtime predictions, which, as we

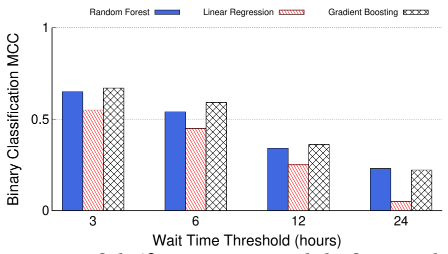


Figure 6: ***MCC of different ML models for predicting job waiting time for different waiting time thresholds*** b.

discuss above, are often not available. In addition, for "optimal" scheduling policies, such as SJF, these approaches also depend on future job arrivals that are not represented when simulating a schedule. While, in this case, ML-based and other statistical approaches suffer from the same limitation, waiting policies can mitigate the difference in waiting time between SJF and FCFS scheduling, which can motivate the use of a simpler scheduling policy like FCFS. Below, we focus on estimating wait times using FCFS scheduling, which are only dependent on the jobs currently in the system.

Rather than rely on job runtime predictions, we instead train a ML model to predict job waiting time based on cluster state, including the number of queued and running jobs, average size of queued and running jobs, average time of running jobs, etc. Our intuition is simple and derives from the law of large numbers: an ML model for predicting job wait time should be more accurate than for predicting job runtime, especially for large clusters, as the former depends on the average runtime of a large number of jobs, while the latter depends on a single job. While any single job's runtime represents just one sample from a workload's job runtime distribution, a large number of queued jobs represents a much larger sample and thus their average job runtime is more likely to be closer to the mean of the job runtime distribution, which determines, in part, a job's wait time.

We use the intuition above to train three different ML models using the features in Table 1, namely linear regression, random forest, and gradient boosting. We simulate execution of the batch trace on our cluster under SWW using our baseline parameters, i.e., b=24h, s=150 `m5.16xlarge` VMs, with work-conserving, non-preemptive FCFS scheduling. For each job, we then record the features from Table 1 at submission time, and then record its waiting time once it is scheduled. Note that *runq-mean-time* and *waitq-mean-time* are jobs' running and waiting times, respectively, up to the present, and so jobs' final running and waiting time may be longer. While our problem is a binary classification, i.e., is the waiting time longer than b, we train multiple regression models using these features to avoid re-training for new values of b.

Figure 6 plots the MCC of our binary classification on waiting time for our baseline b=24h under the different regression models for different values of b. The graph shows

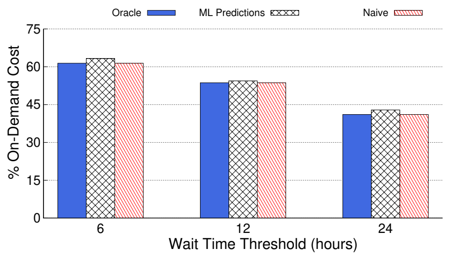


Figure 7: ***On-demand cost, as a percentage of fixed resource cost, for different approaches to predicting job waiting time under SWW with different thresholds*** b.

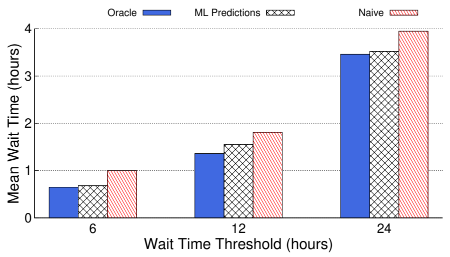


Figure 8: ***Average job waiting time for different approaches to predicting job waiting time under SWW with different thresholds*** b.

that random forest and gradiant boosting have MCCs 0.25-0.7 with the MCCs increasing as the threshold b decreases. By contrast, a naïve approach that forces all jobs to wait yields an MCC of 0. Note that an approach that estimates job waiting time by simulating the schedule forward using our job runtime prediction model from §3 behaves similarly to such a naïve all-jobs-wait approach, since it tends to underpredict each job's waiting time.

We integrated the random forest model above into our simulator to predict job waiting times, and compared its performance both to using an oracle with perfect knowledge of job waiting times, and to the naïve approach above. Figure 7 shows the results for different values of the SWW threshold b along the x-axis, and the additional on-demand cost, as a percentage of the cost of fixed resources, on the y-axis. Here, we again use s=150 `m5.16xlarge` VMs as the number of fixed resources. The graph shows that our ML-based model yields a cost within 2% of the oracle at our baseline b=24h. Note that the naïve approach yields the same cost as the oracle, by definition, but has a mean waiting time that is 14% higher than the oracle at our baseline b=24h, as shown in Figure 8. In contrast, our ML-based waiting time predictions have a waiting time much closer to oracle, >1% at our 24h baseline, and essentially equal at b=6h.

Note that, in practice, to maintain high prediction quality, we would need to periodically re-train our ML models, which is common when deploying ML models in production. In this paper, we also assume our ML models above are trained for a particular workload, and not directly generalizable to other workloads. Designing a transfer learning approach that can be applied to any workload is beyond the scope of this paper.

4 IMPLEMENTATION

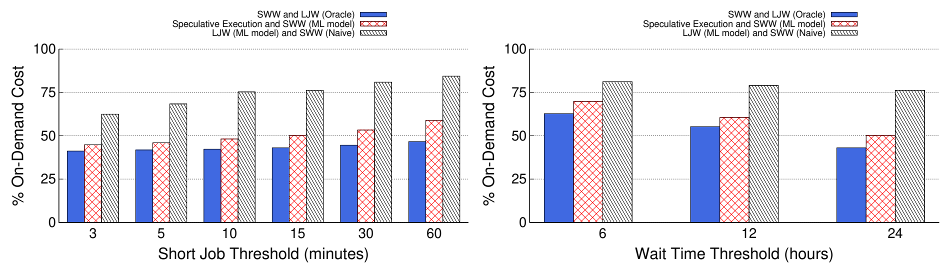
We wrote a trace-driven cloud-enabled job scheduling simulator in python. The simulator enables us to specify the number of fixed resources in the form of VMs with specified cores and memory. We assume jobs' core and memory requirements are rigid, and thus our scheduler cannot adjust them to improve performance. The simulator supports either work-conserving SJF or FCFS scheduling, and the LJW and SWW waiting policies. We have made this simulator publicly available to enable reproducibility [7]. As mentioned earlier, if the waiting policy dictates that a job should run on on-demand resources, the simulator selects the smallest (and cheapest) VMs within EC2's m5 family that satisfies its core and memory requirements. The simulator includes the current per-time price for each of these VMs, which is roughly a linear function of core/memory size. For SJF, LJW, and SWW policies, the simulator uses an API to fetch job runtime and waiting time from a model. We can specify whether this model is an oracle, or one of the ML models from the previous section. We can also specify the short job threshold t (for LJW) and waiting time threshold b (for SWW) at startup. The simulator tracks statistics including average job waiting time, on-demand cost, and average fixed resource utilization.

Implementing waiting policies and our techniques, in practice, is straightforward in batch schedulers, such as Slurm, which already support auto-scaling. Our simulator above does not capture various systems overheads that might occur in a real deployment, such as data staging, resource interference, and fixed delays to boot up VMs, which may be important to scheduling performance in certain contexts. However, these system overheads vary for different configurations of cloud-enabled schedulers, such as cloud bursting from an on-premises cluster or running a native cloud deployment. Evaluating waiting policy performance in these different contexts is outside the scope of this paper. Instead, we focus narrowly on the fundamental cost-waiting time tradeoff exposed by waiting policies.

Our evaluation focuses on two large-scale traces that are representative of job scheduling in academia and industry. Our academic job trace, which we describe in §2.3, is from a shared cluster from a large university system that covers multiple campuses, and thus encompasses the full spectrum of jobs submitted by the medical, science, and engineering research communities. We have batch traces from the cluster's LSF scheduler for the past year, which includes 14 million job submissions; the current cluster's size is 14.3k cores. As we discuss earlier, the trace provides limited information on each job, specifically its submission time, user ID, maximum running time limit, requested number of cores and memory, completion status (finished, terminated, or cancelled), and running time. We have publicly released this job trace at the UMass Trace Repository [7, 8]. Our industry trace is an updated release of the widely-used and publicly-available Google cluster trace, and is an order of magnitude larger [36]. Google uses the Borg scheduler, and we use a portion of the trace that includes 58 million jobs over one week run on a single Borg cell. Note that the Borg scheduler manages both batch and service jobs, where the latter are resource requests for interactive services which typically cannot be arbitrarily delayed [39]. However, since the Google trace does not specify the type of job, we treat them all as delay-tolerant.

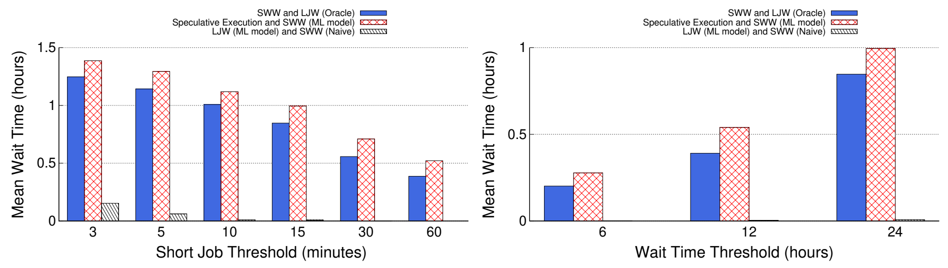
Both of the traces above exhibit the characteristic that a large fraction of jobs are "short" and most of the computation comes from a small set of long jobs. Both characteristics, as we discuss in §3.1, are beneficial for speculative execution. The skew towards short jobs is a common attribute in batch workloads, although the degree of skew varies between workloads [14]. For example, recent work analyzes three other large-scale batch workloads and compares them to an older Google trace [30]. The analysis shows that all traces had a significant skew towards short jobs, although it varies. For example, 65% of Google workload's jobs were less than 6 minutes, while 29%-40% of the three other traces' jobs were less than 6 minutes. Similarly, all traces had a small fraction of long jobs—1%-10% of jobs were greater than 10 hours in duration—-which contribute the majority of cycles compared to the short jobs. Specifically, if we assume jobs have equal resources, a single 10-hour job consumes the same cycles as 100 6-minute jobs, yet there are not 100× more 6-minute jobs. Given the similarity in characteristics between these other batch traces and our industry and academic traces, we expect the general insights and tradeoffs from our evaluation to also apply to these traces (and others with similar characteristics). Of course, the specific thresholds, ML accuracy, costs, and waiting times will vary with the workload.

We reference a number of ML models in both the previous and next section, which we have trained using the traces in conjunction with our simulator. We use python's scikit-learn [16] module for training, and focus on basic models. Our job runtime prediction models are directly trained from the features known at submission time in the trace data above. For our waiting time predictions, we use our simulator to generate a new trace that records the cluster state from Table 1 at each job submission, and then records the job's waiting time once it is scheduled. Of course, this waiting time in the generated dataset is dependent on the number of fixed resources s we configure for our simulator. To generate this new dataset, we use a work-conserving, non-preemptive FCFS scheduling policy with LJW and SWW using our baseline parameters from §2.3. We use this as training data to learn our models of job waiting time. For training our job



(a) (b)

Figure 9: ***On-demand cost, as a percentage of fixed resource cost, on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) using our baseline parameters.***



(a) (b)

Figure 10: ***Mean wait time (hours) on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) using our baseline parameters.***

waiting time ML model, we use 70% of the dataset and 30% of the dataset for testing the models. In addition, we use simple hyperparameters for tuning our ML models, specifically a tree depth of 114 and a random seed of 137 for both random forest and gradient boosting trees.

5 EVALUATION

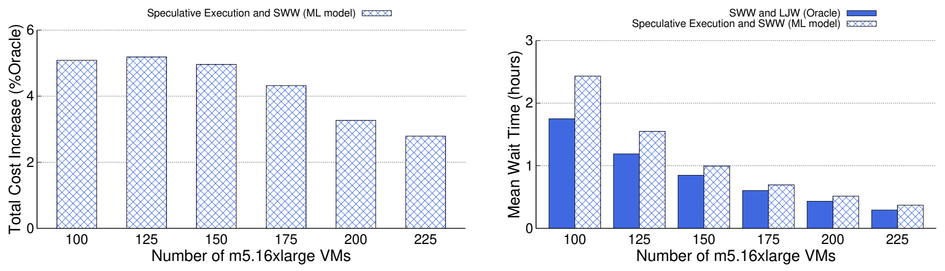
Our evaluation focuses on i) combining the techniques from §3 to quantify how close the cost and waiting time come in practice to that of an oracle; ii) quantifying the effect of the number of fixed resources s on the magnitude of the results; and iii) showing that these techniques also generalize to the Google trace, which has similar job runtime characteristics.

5.1 Combining Techniques

Figure 9 shows the on-demand cost, as a percentage of fixed resource cost, on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) using our baseline parameters. We compare three techniques: SWW and LJW using an oracle with perfect knowledge of job waiting and running time; a naïve approach that uses an ML model for predicting job runtimes for LJW and SWW; and our techniques from §3 that use speculative execution and an ML-based waiting time prediction model. In both graphs, our techniques come much closer to the cost of the oracle compared to those using predictions of job runtime across all short job thresholds t and waiting time thresholds b. Specifically, at our baseline parameters of (t=15m, b=24h) the combined technique comes within 4% of the oracle's on-demand cost. By comparison, using job runtime predictions has a 70% higher cost compared to the oracle. The cost advantage is similar across all parameter settings.

Figure 10 similarly shows the mean wait time on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) using our baseline parameters. This is the same experiment as in Figure 9. Again, combining our techniques from §3 of speculative execution and ML-based waiting time predictions results in a waiting time near that of the oracle across all short job thresholds t and waiting time thresholds b. For our baseline parameters (t=15m, b=24h) combining our techniques comes within 13% of the oracle's mean waiting time. In contrast, a policy that directly uses job runtime predictions for LJW and SWW has nearly zero waiting time because it tends to under-predict job running time due to the large number of short jobs. As a result, it runs most jobs on on-demand resources at a high cost, but with low waiting time.

Recall from §2.3, that the waiting time on the current fixed-size cluster (equivalent to 225 `m5.16xlarge`'s using SJF with perfect knowledge of job running time is 0.6 hours, but is 1.71 hours in practice, when using our job runtime prediction model from §3. For this experiment, the average waiting time across all the parameters are less than 1.71 hours, and many



(a) Cost (b) Wait time

Figure 11: ***Total cost of amortized fixed and on-demand resources (as a percentage of oracle) as a function of fixed resource capacity (a). Mean wait time as a function of fixed resource capacity for our approach and the oracle (b).***

are less than 0.6 hours. Of course, the maximum waiting time in our case is bounded by the waiting time threshold b, while the maximum waiting time is unbounded under SJF. Recall also that the total cost, including both fixed and on-demand resources, of using LJW and SWW under an oracle with our baseline parameters is 5% less than the cost of current fix-sized cluster, and our practical approach achieves near this cost. This shows how optimizing waiting policies for cloud-enabled schedulers can mitigate some of the challenges with optimizing scheduling policies.

Key Point: *Optimizing waiting policies in cloud-enabled schedulers can offset challenges with using optimal scheduling policies, such as SJF.*

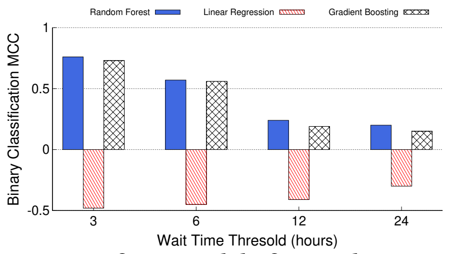


Figure 12: ***MCC of ML models for predicting wait time for various waiting thresholds*** b ***in the Google trace.***

Key Point: *Both the cost (a) and mean waiting time (b) under LJW and SWW using our techniques is near that of an oracle across a wide range of fixed resources.*

5.2 Varying Fixed Resources

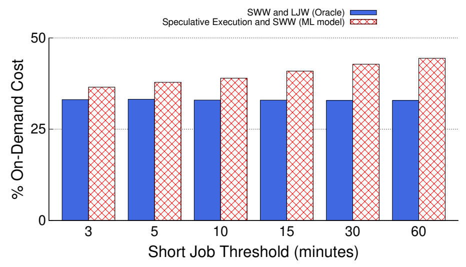
Up to this point, all of our experiments have used the same number of fixed resources s of 150 `m5.16xlarge` VMs, which is optimal number of fixed resources for our workload that minimizes the total cost of fixed and on-demand resources, when amortized over the workload's year-long duration. In this case, we assume the cost of fixed resources is equivalent to the price of 3-year reserved `m5.16xlarge` VM. Figure 11 shows the impact of varying the number of fixed resources on both the cost and waiting time, where are other baseline parameters remain the same. In this case, Figure 11(a) includes the total fixed and on-demand cost for executing the workload under SWW and LJW, as a percentage of the oracle. The graph shows that speculative execution and ML-based waiting time predictions achieves near the same total cost, regardless of number of fixed resources. Note that our cost is closer to the oracle at 150 VMs than above because the previous section only plotted the on-demand cost assuming that fixed resources were a sunk cost.

Figure 11(a) shows that as we increase the number of fixed resources, the average waiting time decreases, as expected, although the percentage difference between our approach and the oracle increases. However, ultimately, the importance of waiting policies decreases as fixed resources increase, since there is less resource constraint and need to wait.

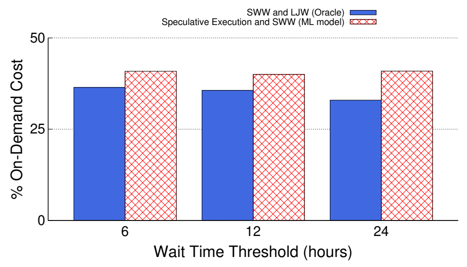
5.3 Generalizing to the Google Workload

Our illustrative examples in §3 and evaluation above are from a single workload. To demonstrate the generality of our approach, we performed a similar evaluation using the Google trace [36]. The trace includes data from 8 Borg cells over a single month in May 2019. Since the number of jobs is massive, we focus on a single week from a single cell, which includes 58 million job submissions. We further randomly sample this down to 14 million job submissions, or 25%, to reduce the overhead of our simulations. Note that our sampled trace has the same mean core/memory request and job runtime as the original. The Google trace's jobs have much larger core/memory requirements than our academic trace, so we adjust the number and size of our baseline fixed resources for this evaluation. We set our baseline fixed resources s=4000 VMs, each with 192 cores and 768GB memory.

We use the same per-core pricing as in the previous experiments, which is based on the `m5` family, i.e., \$0.048 per core-hour. The `N2` family of VMs in Google Compute Engine (GCE) have a similar price. In this case, for on-demand VMs, we assume the use of custom VMs from GCE with \$0.031611 per core-hour and \$0.004237 per GB-hour, as the Google trace has many jobs with unbalanced core/memory ratios that waste significant resources when using fixed-size VMs. These custom prices are equivalent to \$0.048 per hour for 1

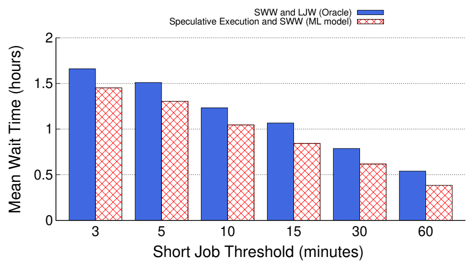


(a)

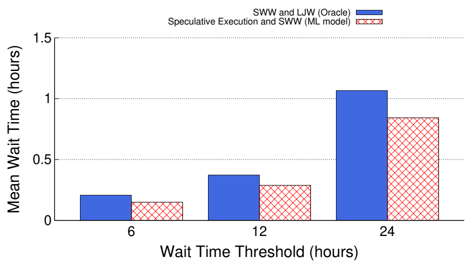


(b)

Figure 13: ***On-demand cost, as a percentage of fixed resource cost, on the y-axis as a function of both LJW's short job threshold*** t ***(a) and SWW's waiting time threshold*** b ***(b) for our Google trace using the baseline parameters.***



(a)



(b)

Figure 14: ***Mean wait time (hours) on the y-axis as a function of both LJW's short job threshold*** t ***(a) and SWW's waiting time threshold*** b ***(b) for Google trace using our baseline parameters.***

core and 4GB memory, as above. As before, we assume the amortized cost of the fixed resources has a 60% lower cost, equivalent to that of a 3-year reserved VM. We use the same baseline parameters as in the other analysis (t=15m, b=24h).

Recall from Figure 3 in §3 that the Google workload's job runtime distribution is remarkably similar in shape to that of our academic batch trace, where a significant fraction of jobs are short, but where much of the computation comes from a small fraction of long jobs. We also trained waiting time prediction models using the same approach as in §3.2 by generating a dataset from a simulation run that recorded the features listed in Table 1. Figure 12 plots the Matthews Correlation Coefficient (MCC) for these models. The results for a random forest and gradiant boosting model are similar to those in our academic workload, from Figure 6, while linear regression actually exhibits a negative MCC. We use a random forest model, since it yields the highest MCC.

Figure 13 shows the on-demand cost as a percentage of fixed resource cost (on the y-axis) as a function of LJW's short job threshold t (a) and SWW's waiting time threshold b (b) for our Google trace using the baseline parameters. Figure 13(a) shows the same trends as in Figure 9(a) for speculative execution as the short job threshold t changes. For small values of t, the difference in cost between the oracle and speculative execution is minimal because a large fraction of jobs are short, and thus *should* run on on-demand VMs. Similarly, Figure 13(b) also shows the same trend as in Figure 9(b) as the wait time threshold b changes, where slightly shorter thresholds have a cost closer to the oracle. Again, our observation that the accuracy of waiting time predictions based on cluster state is aided by the law of large numbers is general, and also holds for the Google trace.

Figure 14(b) shows the average waiting time for the same experiment as above. We see similar trends as in Figure 10(b) using the academic workload, except that the waiting time for our approach in the Google trace is actually slightly less than when using an oracle. This occurs because our ML-based waiting time prediction model performs slightly better compared to these models for our academic trace. As a result, there are fewer jobs that actually end up waiting for fixed resources for time b in the queue, and then also incur the high price of using on-demand VMs. These waiting time mis-predictions are the reason our academic batch workload has both a slightly higher cost and waiting time compared to the oracle. When using an oracle, cost and waiting time are a tradeoff: using more on-demand VMs incurs a higher cost, but should lower waiting time, since there is no reason to wait for an on-demand VM with an oracle. However, mis-predictions can cause waiting for on-demand VMs in practice, which can increase average waiting time. This happens much less in the Google trace compared to the academic batch workload, and thus higher cost compared to the oracle is compensated by a lower average waiting time.

Key Point: *The Google workload exhibits similar trends as our academic workload.*

6 RELATED WORK

Conventional job scheduling on fixed resources has been studied for decades, and continues to be an active area of research [23, 29, 38]. Prior work has examined the problem in many contexts, e.g., with deadlines, priorities, fairness constraints, etc. As more computation shifts to cloud platforms, conventional job scheduling is becoming less important for cloud users, since clouds provide the illusion of infinite scalability. While clouds are, of course, not infinitely scalable, they are sufficiently large now that users generally never experience any resource constraint. Without any resource constraint, cloud users no longer have a scheduling problem, but instead have a cost problem. Of course, cloud platforms must still address conventional scheduling [39]. However, there are many more cloud users than cloud platforms.

To lower their cost, cloud users have an incentive to buy some fixed resources upfront, since they cost less than on-demand resources as long as they are highly utilized. However, jobs need not wait for fixed resources when fully utilized, since they can always run on on-demand resources. This dynamic introduces the waiting problem, as schedulers must now decide which jobs should wait and for how long. Recent work introduces the waiting problem for cloud-enabled schedulers, and analyzes it using a $M/M/s$ queuing model [13]. However, that work focuses on optimizing fixed resource provisioning to minimize cost assuming that the waiting policies had perfect knowledge of job running and waiting times. We show how to realize these waiting policies in practice without perfect knowledge using speculative execution and ML-based waiting time predictions.

Despite its importance to cloud scheduling, we have not seen any other prior work that directly addresses waiting policies. Our work is related to prior work on scheduling for hybrid clouds, which include fixed private resources, but can also burst into the cloud [21, 26]. However, that work does not define the notion of a waiting policy. In some sense, cloud schedulers that "auto-scale" by dynamically adding resources to service jobs implement an implicit waiting policy, where jobs never wait [4]. Auto-scalers often increase a cluster's size when demand increases to maintain some SLO threshold [4]. However, auto-scaling policies are orthogonal to waiting policies. For example, an auto-scaler might increase cluster size whenever the mean waiting time from a waiting policy exceeds a SLO threshold. Thus, auto-scaling and waiting policies may work in concert with each other.

The focus of our work is to demonstrate that we can realize waiting policies in practice that are close to optimal, given *a priori* knowledge of job running and waiting time. There has been substantial prior work on predicting job running and waiting time for cluster job schedulers, although much of it is not used in practice [17, 18, 20, 28, 37]. For example, [37] uses a clustering approach that groups jobs by their attributes and then predicts job runtime within each group, while [18] and [19] provide upper bounds on job and workload duration based on prior prediction errors. Recent work details the many reasons why cluster schedulers do not use job running time predictions [25], including low accuracy due to insufficient data, non-stationarity, and unfair performance. Our work echoes many of the same points, as standard ML models cannot even accurately categorize job running times to be above or below a threshold.

There is much less work on predicting job waiting time because it does not directly benefit conventional scheduling [15, 33]. Prior work generally builds on job runtime predictions rather than using cluster state. None of this work applies their prediction methods to waiting policies, but instead looks at other scenarios where jobs have a choice among multiple queues or can modify their request to reduce waiting time. Our focus is not necessarily on developing the most accurate waiting time prediction model, but instead to show that basic models can do well in the context of waiting policies largely due to the law of large numbers.

Finally, there has also been significant recent work on leveraging cheap spot/preemptible VMs to lower the cost of running jobs in the cloud [22, 31, 35, 40, 41]. Much of this work focuses on data-parallel jobs [31, 40, 41], rather than batch scheduling. However, prior work has observed that the longer jobs run on spot/preemptible VMs, the more likely they are to be revoked [32]. Spot/preemptible VMs are more well-suited for speculative execution, which only runs jobs for a short period of time and thus are less susceptible to revocations. We could also replace on-demand VMs with spot/preemptible VMs (with checkpointing) to lower costs. However, while the magnitude of costs might change, our primary observations and results on waiting policies would not change.

7 CONCLUSION

This paper focuses on optimizing the cost-waiting time tradeoff for cloud-enabled schedulers, which can run jobs on either fixed or on-demand resources. This tradeoff is dependent on the scheduler's *waiting policy*, and optimizing the waiting policy generally requires *a priori* knowledge of job runtime. We present two techniques—speculative execution and ML-based waiting time predictions—that enable implementing near-optimal waiting policies in practice without accurate job runtime predictions. We evaluate these techniques on two large job traces from academia and industry, and show they yield a cost and waiting time near that of an oracle with perfect knowledge of job running and waiting time.

Acknowledgements. This work is funded by AWS and NSF grants CNS-1925464, CNS-1908536, and CNS-2105494.

REFERENCES

[1] 2019. Slurm Elastic Computing (Cloud Bursting). https://slurm.schedmd.com/elastic_computing.html.

[2] 2019. Slurm Workload Manager. https://slurm.schedmd.com/.

[3] 2020. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[4] 2020. AWS ParallelCluster Auto Scaling. https://docs.aws.amazon.com/parallelcluster/latest/ug/autoscaling.html.

[5] 2020. Azure Spot Virtual Machines. https://azure.microsoft.com/en-us/pricing/spot/.

[6] 2020. Google Preemptible Virtual Machines. https://cloud.google.com/preemptible-vms.

[7] 2020. UMass Trace Repository. http://traces.cs.umass.edu/.

[8] 2020. Waiting Game Job Trace. https://doi.org/10.5281/zenodo.3872168.

[9] 2021. AWS Batch - Fully managed batch processing at any scale. https://aws.amazon.com/batch/.

[10] 2021. Azure Batch - Cloud-scale job scheduling and compute management. https://azure.microsoft.com/en-us/services/batch/.

[11] 2021. Load Sharing Facility. https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=lsf-foundations.

[12] O. Alipourfard, H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*.

[13] P. Ambati, N. Bashir, D. Irwin, and P. Shenoy. 2020. Waiting Game: Optimally Provisioning Fixed Resources for Cloud-Enabled Schedulers. In *SC*.

[14] G. Amvrosiadis, J.W. Park, G. Ganger, G. Gibson, E. Baseman, and N. DeBardeleben. 2017. *Bigger, Longer, Fewer: What Do Cluster Jobs Look Like Outside Google?* Technical Report CMU-PDL-17-104.

[15] J. Brevik, D. Nurmi, and R. Wolski. 2006. Predicting Bounds on Queuing Delay for Batch-Scheduled Parallel Machines. In *PPoPP*.

[16] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API Design for Machine Learning Software: Experiences from the cikit-learn Project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[17] S. Di, D. Kondo, and C. Wang. 2013. Optimization and Stabilization of Composite Service Processing in a Cloud System. In *2013 IEEE/ACM 21st International Symposium on Quality of Service (IWQoS)*.

[18] S. Di, C. Wang, and F. Cappello. 2014. Adaptive Algorithm for Minimizing Cloud Task Length with Prediction Errors. *IEEE Transactions on Cloud Computing* 2, 2 (2014), 194–207. https://doi.org/10.1109/TCC.2013.16

[19] S. Di, C. Wang, D. Kondo, and G. Han. 2013. Towards Payment-Bound Analysis in Cloud Systems with Task-Prediction Errors. In *2013 IEEE Sixth International Conference on Cloud Computing*.

[20] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*.

[21] T. Guo, U. Sharma, S. Sahu, T. Wood, and P. Shenoy. 2012. Seagull: Intelligent Cloud Bursting for Enterprise Applications. In *USENIX ATC*.

[22] A. Harlap, A. Tumanov, A. Chung, G. Ganger, and P. Gibbons. 2017. Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets. In *European Conference on Computer Systems (EuroSys)*.

[23] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*.

[24] J. Kadupitige, V. Jadhao, and P. Sharma. 2020. Modeling the Temporally Constrained Preemptions of Transient Cloud VMs. In *HPDC*.

[25] Michael Kuchnik, J. Park, C. Cranor, Elisabeth Moore, Nathan DeBardeleben, and George Amvrosiadis. 2019. *This is Why ML-driven Cluster Scheduling Remains Widely Impractical.* Technical Report CMU-PDL-19-103.

[26] S. Niu, J. Zhai, X. Ma, X. Tang, and W. Chen. 2013. Cost-effective Cloud HPC Resource Provisioning by Building Semi-Elastic Virtual Clusters. In *SC*.

[27] D. Nurmi, J. Brevik, and R. Wolski. 2007. QBETS: Queue Bounds Estimation from Time Series. In *JSSPP*.

[28] S. Omer, N.Yigitbasi, A. Iosup, and D. Epema. 2009. Trace-based Evaluation of Job Runtime and Queue Wait Time Predictions in Grids. In *HPDC*.

[29] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 2018. 3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*. https://doi.org/10.1145/3190508.3190515

[30] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2011. *Google cluster-usage traces: format + schema.* Technical Report. Google Inc., Mountain View, CA, USA. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[31] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. 2016. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *European Conference on Computer Systems (EuroSys)*.

[32] S. Shastri, A. Rizk, and D. Irwin. 2016. Transient Guarantees: Maximizing the Value of Idle Cloud Capacity. In *SC*.

[33] W. Smith, V. Taylor, and I. Foster. 1999. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In *JSSPP*.

[34] Abel Souza, Kristiaan Pelckmans, Devarshi Ghoshal, Lavanya Ramakrishnan, and Johan Tordsson. 2020. ASA - The Adaptive Scheduling Architecture. In *HPDC*.

[35] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. 2015. SpotOn: A Batch Computing Service for the Spot Market. In *Symposium on Cloud Computing (SoCC)*.

[36] M. Tirmazi, A. Barker, N. Deng, M. Haque, Z. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. 2020. Borg: The Next Generation. In *EuroSys*.

[37] A. Tumanov, A. Jiang, J. Park, M. Kozuch, and G. Ganger. 2016. JamaisVu: Robust Scheduling with Auto-Estimated Job Runtimes.

[38] A. Tumanov, T. Zhu, J. Park, M. Kozuch, M. Harchol-Balter, and G. Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *EuroSys*.

[39] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *European Conference on Computer Systems (EuroSys)*.

[40] Y. Yan, Y. Gao, Z. Guo, B. Chen, and T. Moscibroda. 2016. TR-Spark: Transient Computing for Big Data Analytics. In *Symposium on Cloud Computing (SoCC)*.

[41] Y. Yang, G. Kim, W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B. Chun. 2017. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *European Conference on Computer Systems (EuroSys)*.