Jockey: Guaranteed Job Latency in Data Parallel Clusters

Andrew D. Ferguson
Brown University
adf@cs.brown.edu

Peter Bodik
Microsoft Research
peterb@microsoft.com

Srikanth Kandula
Microsoft Research
srikanth@microsoft.com

Eric Boutin
Microsoft Bing
eric.boutin@microsoft.com

Rodrigo Fonseca
Brown University
rfonseca@cs.brown.edu

Abstract

Data processing frameworks such as MapReduce [8] and Dryad [11] are used today in business environments where customers expect guaranteed performance. To date, however, these systems are not capable of providing guarantees on job latency because scheduling policies are based on fair-sharing, and operators seek high cluster use through statistical multiplexing and over-subscription. With Jockey, we provide latency SLOs for data parallel jobs written in SCOPE. Jockey precomputes statistics using a simulator that captures the job's complex internal dependencies, accurately and efficiently predicting the remaining run time at different resource allocations and in different stages of the job. Our control policy monitors a job's performance, and dynamically adjusts resource allocation in the shared cluster in order to maximize the job's economic utility while minimizing its impact on the rest of the cluster. In our experiments in Microsoft's production Cosmos clusters, Jockey meets the specified job latency SLOs and responds to changes in cluster conditions.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling

General Terms Algorithms, Performance

Keywords deadline, scheduling, SLO, data parallel, dynamic adaptation, Dryad, MapReduce

1. Introduction

Batch processing frameworks for data parallel clusters such as MapReduce [8] and SCOPE [6] on Dryad [11] are seeing increasing use in business environments as part of near-real time production systems at Facebook [5] and Microsoft. These frameworks now run recurring, business-critical jobs, and organizations require strict service-level objectives (SLOs) on latency, such as finishing in less than one hour. Missing a deadline often has significant consequences for the business (e.g., delays in updating website content), and can result in financial penalties to third parties. The outputs of many jobs feed into other data pipelines throughout the company; long job delays can thus affect other teams unable to fix the input jobs. Operators who monitor these critical jobs are alerted when they fall behind, and have to manually resolve problems by restarting jobs, or adjusting resource allocations. A framework which automatically provided latency SLOs would eliminate such manual repairs.

The ability to meet an SLO in data parallel frameworks is challenging for several reasons. First, unlike interactive web requests [23], data parallel jobs have complex internal structure with operations (e.g., map, reduce, join, etc.) which feed data from one to the other [6, 7]. Barriers, such as aggregation operations, require the synchronization of all nodes before progress can continue. Failures, be they at task, server or network granularity, cause unpredictable variation, and particularly delay progress when they occur before a barrier.

Secondly, statistical multiplexing and over-subscription ensure high utilization of such clusters. This creates variability in response times due to work performed by other jobs. Finally, work-conserving allocation policies add variation by providing jobs with spare resources [12, 27]. Under these policies, each admitted job is guaranteed some task slots; slots that go unused are distributed to other jobs that have pending tasks. While this improves cluster efficiency, job latency varies with the availability of spare capacity in the cluster.

We provide latency guarantees for data parallel jobs in shared clusters with Jockey, which combines a detailed per-job resource model with a robust control policy. Given a pre-

EuroSys'12, April 10–13, 2012, Bern, Switzerland.
Copyright © 2012 ACM 978-1-4503-1223-3/12/04. . . $10.00

vious execution of the job[1] and a utility function, Jockey models the relationship between resource allocation and expected job utility. During job runtime, the control policy computes the progress of the job and estimates the resource allocation that maximizes job utility and minimizes cluster impact by considering the task dependency structure, individual task latencies, and failure probabilities and effects.

While the resource allocator in Jockey operates on individual jobs, we can use admission control to ensure that sufficient guaranteed capacity is available to all admitted SLO jobs. Jockey's job model can be used to check whether a newly submitted job would "fit" in the cluster – that is, that all previously accepted SLO jobs would still be able to meet their deadlines – before permitting it to run. If a submitted SLO job does not fit in the cluster, the cluster scheduler would need to arbitrate between the jobs to determine an allocation which maximizes the global utility at the risk of missing some SLO deadlines. We leave the development of such a global arbiter as future work.

Prior approaches to providing guaranteed performance fall into one of three classes. The first class partitions clusters into disjoint subsets and is used at companies such as Facebook [10]. Jobs which require guaranteed performance are run in a dedicated cluster, and admission control prevents contention between jobs. This class achieves guarantees by sacrificing efficiency because the dedicated cluster must be mostly idle to meet SLOs. A second class of solutions shares the cluster, but provides priority access to SLO-bound jobs – tasks from such jobs run when ready and with optimal network placement. This shields SLO-bound jobs from variance due to other jobs. However, the impact on non-SLO jobs is significant: their partially complete tasks may have to vacate resources or lose locality when a higher-priority task arrives. In addition, this approach can only support a limited number of SLO-bound jobs to prevent negative interference between them. A final class of solutions, common across many domains, models the workload and selects a static resource allocation that ensures the deadline is met. We find that simple models for more general data parallel pipelines are imprecise, and dynamic adaptation is necessary to cope with runtime changes in the cluster and job structure.

Our core contribution is an approach that combines a detailed job model with dynamic control. Experiments on large-scale production clusters indicate that Jockey is remarkably effective at guaranteeing job latency – in 94 experiments it missed only a single deadline, by only 3% – and that neither the model nor control is effective without the other. Jockey is successful because it (a) minimizes the impact of SLO-bound jobs on the cluster while still providing guarantees, (b) pessimistically over-allocates resources at the start to compensate for potential future failures, and (c) can meet latency SLOs without requiring guaranteed performance from individual resources such as the cluster network and disks.

[1] Recurring jobs, which include most SLO-bound jobs, account for over 40% of runs in our cluster, providing ready historical data for our models.

2. Experiences from production clusters

To motivate our method for guaranteeing job latency in a production data-parallel cluster, we first describe the architecture of the cluster and the importance of latency SLOs. We then show that SLOs are difficult to meet due to high variance in job latency, and illustrate the causes of such variance.

2.1 Cluster Background

To gain insight into the problem, we examine a single cluster in Cosmos, the data parallel clusters that back Bing and other Microsoft online properties. Example applications running on this cluster include generating web indices, processing end-user clickstreams, and determining advertising selections. Jobs are written in SCOPE [6], a mash-up language with both declarative and imperative elements similar to Pig [17] or HIVE [22]. A compiler translates the job into an *execution plan graph* wherein nodes represent *stages* such as map, reduce or join, and edges represent dataflow [7, 9, 12]. Each stage consists of one or more parallel *tasks*. For stages that are connected by an edge, communication between their tasks ranges from one-to-one to all-to-all. A *barrier* occurs when tasks in a dependent stage cannot begin until every task in the input stage finishes. Barriers are often due to operations that are neither associative nor commutative. Job data files reside in a distributed file system which is implemented using the same servers that run tasks, similar to Hadoop's HDFS or the Google File System [9]. The cluster is shared across many business groups; at any time, there are many jobs running in the cluster and several tasks running on each server.

Similar to other cluster schedulers [27], our cluster employs a form of fair sharing across business groups and their jobs. Each job is guaranteed a number of *tokens*, as dictated by cluster policy, and each running task uses one token, which is released upon task completion. For efficiency, spare tokens are allocated to jobs that have pending tasks. Jobs are admitted to the cluster such that the total tokens guaranteed to admitted jobs remains bounded. While a token guarantees a task's share of CPU and memory, other resources such as network bandwidth and disk queue priority are left to their default sharing mechanisms, which are either per-flow or per-request based.

2.2 SLOs in Data Parallel Clusters

Setting an SLO deadline depends on a number of factors, most of which relate to the job's purpose. At a minimum, the deadline must be feasible: it cannot be shorter than the amount of time required to finish the job given an infinite amount of resources (*i.e.*, the length of the critical path). Feasibility can be checked with trial job executions, or estimated using a simulator such as the one in Jockey (see Section 4.1).

Deadlines for some jobs are derived from contractual agreements with external (non-Microsoft) customers, such as advertisers or business partners, while others are set to ensure customer-facing online content is kept fresh and up-to-date. In each case, missing a deadline can be financially detrimental to the business, either because of a contractually-specified penalty or the associated loss of revenue. Because final outputs are often the product of a pipeline of jobs, a deadline on the final output leads to individual deadlines for many different jobs running in Cosmos.

Finally, many internal deadlines are "soft" – that is, finishing after four hours instead of three is undesirable, but does not trigger a financial penalty. However, a single cluster runs a large number of concurrent jobs, some of which have no deadlines, some have soft deadlines, and some have very strict deadlines. With standard weighted fair sharing, it is difficult to map latency objectives for each of type of deadline onto an appropriate weight. Directly specifying a utility function to indicate a job's deadline and importance alleviates this problem for our users.

In our experiments (Section 5), we set the target deadline based on the length of the critical path, and for seven of the jobs, we test with two different deadlines.

2.3 Variance in Job Latency

We quantify variance in the cluster by comparing completion times across runs of recurring jobs, such as the many production jobs which repeatedly execute on newly arrived data. By being mostly similar, recurring jobs provide a ready yet real source for cross-job comparison. The executions we examine consist of production-cluster jobs that repeated at least ten times each during September 2011.

Across the runs of each recurring job, we compute the completion time's coefficient of variation (CoV), i.e., $\frac{stdev}{mean}$. Table 1 shows that the median recurring job has a CoV of 0.28, and 10% of all jobs have a CoV over 0.59. While a CoV value less than 1 is considered to be low variance, these results imply that for half (or 10%) of recurring jobs the latency of a sixth of their runs is > 28% (or > 59%) larger than the mean.

We find that the size of the input data to be processed varies across runs of recurring jobs. To discount the impact of input size on job latency, we further group runs of the same job into clusters containing runs with input size differing by at most 10%. Table 1 shows that much of the variation still persists even within these clusters.

2.4 Causes of Variance

A potential cause of job latency variance is the use of spare tokens. Recall that our cluster re-allocates tokens which are unused by the jobs to which they were guaranteed. To explore this hypothesis, we compared runs of seven jobs described in Section 5.2 with experimental runs that were restricted to using guaranteed capacity only – the CoV dropped by up to five times. While these jobs are smaller than the median job in our cluster, and thus the specific decrease may not be representative, we believe it confirms our hypothesis that spare tokens add variance to job latency.

Statistic	Percentiles			
	10th	50th	90th	99th
CoV across recurring jobs	.15	.28	.59	1.55
CoV across runs with inputs differing by at most 10%	.13	.20	.37	.85

Table 1. The coefficient of variation (CoV) of completion time across runs of recurring jobs. Variation persists across runs with similar input sizes.

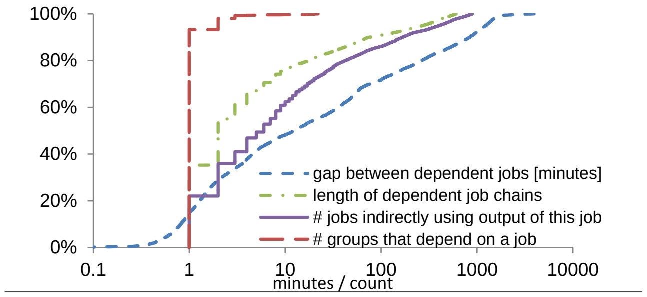


Figure 1. Dependence between jobs: 20% of jobs have more than 20 other jobs depending on their output. Over half of the directly dependent jobs start within 10 minutes of the earlier job and are hence likely to stall if the earlier job is delayed. Long chains of dependent jobs are common, and many chains span business groups.

The use of spare capacity creates variance in a job's run time for two reasons. First, the availability of spare tokens fluctuates because it depends on the nature of other jobs running in the cluster – if other jobs have more barriers or more outliers due to data skew, more tokens will be spare. In the above experiments, the fraction of the job's vertices that executed using the spare capacity varied between 5% and 80%. Second, tasks using spare tokens run at a lower priority than those using guaranteed tokens, and thus can be evicted or pushed into the background during periods of contention.

Task runtimes also vary due to hardware and software failures, and contention for network bandwidth and server resources. In public infrastructures, such as EC2 and Azure, such contention is even higher than in our cluster [13, 25].

2.5 Impact on Job Pipelines

Because many business processes consist of pipelines of multiple jobs, variance in the completion time of a single job can have a wide impact. To quantify this impact, we examined all jobs in our cluster over a period of three days. When a job's input contains data blocks written by an earlier job, we infer a dependence. We did not track dependences due to changes to the filesystem (e.g., copying or renaming blocks) and use of data outside the cluster (e.g., downloading a job's output to train a classifier which is then used by other jobs).

For the 10.2% of jobs with at least one dependency, which includes most SLO-bound jobs, Fig. 1 quantifies those dependences. The violet (solid) line shows that the median job's output is used by over ten other jobs – for the top 10% of jobs, there are over a hundred dependent jobs. The blue (small dashes) line shows that many directly dependent jobs start soon after the completion of a job – the median gap is ten minutes. This means that delays in the job will delay the start of these subsequent jobs. The green (dash-dot line) shows that the chains of dependent jobs can be quite long and span different business groups (red or big dash line). At business group or company boundaries, these delays can cause financial penalties and require manual intervention.

2.6 Lessons for Jockey

Jockey uses the number of guaranteed tokens as the mechanism to adjust a job's performance because it directly addresses one source of variance in our cluster. Because our tokens are analogous to tickets in a lottery scheduler or the weights in a weighted fair queuing regime, Jockey's methodology is directly applicable to other systems which use a weighted fair approach to resource allocation.

Jockey uses readily available prior executions to build a model of a recurring job's execution. Such a model is essential to translating resource allocation into expected completion time. We will show later how Jockey makes use of prior executions despite possible variations in input size.

3. Solutions for Providing SLOs

We consider three solutions to our goal of providing SLO-like guarantees of job completion times in Cosmos. The first is to introduce an additional priority class in the cluster-wide scheduler, and map different SLOs onto each class. The second is to manually determine resource quotas for each job. Finally, we develop a novel method to dynamically adjust resources based on the job's current performance and historical data.

3.1 Additional priority classes

The first potential solution is to implement a third class of tokens with a new, higher priority. Jobs with the strictest SLOs can be allocated and guaranteed these "SuperHigh" tokens. Through the combination of strict admission control, repeated job profiling to determine the necessary allocation, and a paucity of SuperHigh tokens at the cluster-scale, it is possible to meet SLOs with this design.

However, there are numerous downsides to this approach. When a job runs with SuperHigh tokens it increases contention for local resources. This has a negative impact on regular jobs, which can be slowed or potentially lose locality – the beneficial co-location of storage and computational resources. Secondly, the cluster scheduler must be overly pessimistic about the number of SuperHigh-priority jobs which can execute simultaneously. If too many such jobs are admitted to the cluster, the jobs will thrash and cluster goodput will fall. Finally, the heart of this solution is to introduce ordinal priority classes into the system, which are known to have weak expressive power and can lead to poor scheduling decisions when the system is overloaded [4]. We did not further evaluate this solution because its use would impact actual SLO-bound jobs in our production cluster.

3.2 Quotas for each job

A second potential solution for meeting SLO-bound jobs is to introduce strict, static quotas with the appropriate number of guaranteed tokens for each job. This solution is evaluated in Section 5.2 as *Jockey w/o adaptation*, and we find it to be unsatisfactory for three reasons. First, as cluster conditions change due to node failures and other events detailed later, the number of tokens required to meet the SLO also changes. Therefore, it would be necessary to regularly rebalance the quotas for all such SLO jobs.

Second, we have observed that determining weights and quotas is difficult for many users of large clusters. To reduce the chance of missing an SLO, some users request too many resources, which makes useful admission control challenging. Others request too few because they have relied on overly-optimistic trial runs, or a tenuous bounty of spare capacity tokens in the past. To explore the ability of users to correctly size their resource requests, we examined the guaranteed allocations and the maximum achieved parallelism of production jobs during a one-month period. We found that the maximum parallelism of one-third of the jobs was less than the guaranteed allocation. Futhermore, the maximum parallelism of one-quarter of the jobs reached more than ten times the guaranteed allocation thanks to the spare capacity.

Finally, it is clear that when multiple SLO-bound jobs exist in the system, the cluster's goodput can be improved by dynamically re-allocating resources from jobs with slack SLOs to those with tight SLOs.[2] This motivates the design of our solution, along with additional requirements described next.

3.3 Dynamic resource management

In order to meet the desired SLOs in Cosmos, we developed a dynamic resource management system, Jockey, which we describe in detail in Section 4. Our design was guided by the limitations of the two solutions above, the variability of job performance described in Section 2.3, and the structure of SCOPE programs. We also faced additional constraints such as the need to adapt to changes in cluster availability, delays in job submission, and changes in the SLO after job initialization.

The variability of job performance implies that the scheduler needs to react to changing cluster conditions, periodically re-allocating resources at a fine timescale during job execution. We discuss the sensitivity of the scheduler to this timescale in Section 5.5. Because resource allocations are recalculated during the job's execution, it is necessary to have an accurate indicator of the job's current progress, in addition to a model of the job's end-to-end latency.

The DAG structure of jobs in Cosmos creates two challenges. A first is that Jockey must make decisions which respect dependencies between tasks. A second is the wide variation in a job's degree of parallelism during execution. Some stages may be split into hundreds of tasks, while others, such

[2] A few Cosmos users even tried to do this by hand in the past!

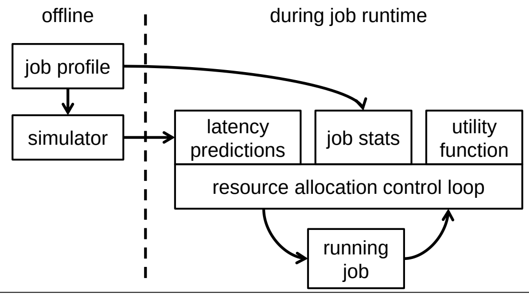


Figure 2. Architecture diagram: in the offline phase, we use a profile of a previous run to estimate job statistics and use the simulator to estimate completion times. During runtime, the control loop monitors the job and uses job statistics, latency predictions and the utility function to propose the minimum resource allocation that maximizes the utility of the job.

as an aggregation stage, are split into few tasks. The scheduler must allocate enough resources early in the job so that it does not attempt in vain to speed-up execution by increasing the resources for a later stage beyond the available parallelism. Jockey must also be aware of the probability and effect of failures at different stages in the job so there is an appropriate amount of time remaining to recover before the deadline.

4. Jockey

Jockey is composed of three components: a *job simulator*, which is used offline to estimate the job completion time given the current job progress and token allocation, a *job progress indicator*, which is used at runtime to characterize the progress of the job, and a *resource allocation control loop*, which uses the job progress indicator and estimates of completion times from the simulator to allocate tokens such that the job's expected utility is maximized and its impact on the cluster is minimized (see the architecture diagram in Fig. 2). We describe these components in more detail in the following three sections, and address limitations of our approach in Section 4.4.

4.1 Job Completion Time Prediction

In order to allocate the appropriate number of tokens to meet an SLO, Jockey must be able to predict the job's completion time under different token allocations given the current progress. This is challenging because the system has to consider all remaining work in the job and the dependencies between stages. We consider two methods for this prediction: an event-based simulator, and an analytical model inspired by Amdahl's Law. Based on our evaluation in Section 5.3, we use the simulator approach in the current version of Jockey.

Job simulator and the offline estimation

The job simulator produces an estimate of the job completion time given a particular allocation of resources and job progress. These estimates are based on one or more previous runs of the job, from which we extract performance statistics such as the per-stage distributions of task runtimes and initialization latencies, and the probabilities of single and multiple task failures. The job simulator takes as input these statistics, along with the job's algebra (list of stages, tasks and their dependencies), and simulates events in the execution of the job. Events include allocating tasks to machines, restarting failed tasks and scheduling tasks as their inputs become available. This simulator captures important features of the job's performance such as outliers (tasks with unusually high latency) and barriers (stages which start only when all tasks in dependent stages have finished), but does not simulate all aspects of the system, such as input size variation and the scheduling of duplicate tasks. We discuss the accuracy of the simulator in Section 5.3.

A basic implementation of the resource allocation control loop could invoke the simulator during each iteration by marking the completed tasks and simulating forward. Then, for each resource allocation under consideration, multiple simulations could be used to estimate the distribution of completion times and thus the expected utility given that allocation. However, depending on the number of allocations considered and the size of the job, these simulations could take a long time and add a significant delay to the control loop. Therefore, we develop a method that only uses the simulator offline, precomputing all information necessary to accurately and quickly allocate resources.

For each SLO job, we estimate $C(p, a)$ – a random variable denoting the remaining time to complete the job when the job has made progress p and is allocated a tokens. In the control loop, we use these precomputed values to select an appropriate allocation. We present an approach to compute the job progress p in Section 4.2.

We estimate the distribution of $C(p, a)$ by repeatedly simulating the job at different allocations. From each simulation, say at allocation a that finishes in time T, we compute for all discrete $t \in [0, T]$ the progress of the job p_t at time t and the remaining time to completion $t_c = T - t$. Clearly, $t_c = C(p_t, a)$, i.e., the value t_c is one sample from the distribution of $C(p_t, a)$. Iterating over all t in a run and simulating the job many times with different values of a provides many more samples, allowing us to estimate the distribution well. Because the logic in the simulator is close to that of the real system, these estimates approximate real run times well.

Amdahl's Law

Rather than using the simulator above, we can use a modified version of Amdahl's Law [1] to estimate the job's completion time given a particular allocation. Amdahl's Law states that if the serial part of a program takes time S to execute on a single processor, and the parallel part takes time P, then running the program with N processors takes $S + P/N$ time. In our case, we let S be the length of the critical path of the job and P be the aggregate CPU time spent executing the job, minus the time on the critical path. To estimate the remaining completion time of a job when allocated a tokens, we evaluate the above formula with $N = a$.

To use Amdahl's Law in our resource allocation loop, we need to estimate the total work remaining in the job, P_t, and the length of the remaining critical path, S_t, while the job is running. For each stage s, let f_s be the fraction of tasks that finished in stage s, l_s be the execution time of the longest task in stage s, L_s be the longest path from stage s to the end of the job and T_s be the total CPU time to execute all tasks in stage s. Note that the last three parameters can be estimated from prior runs before the job starts, and f_s can easily be maintained by the job manager at run time. Now, $S_t = \max_{\text{stage } s: f_s < 1} (1 - f_s) l_s + L_s$ and $P_t = \sum_{\text{stage } s: f_s < 1} (1 - f_s) T_s$. In words, across stages with unfinished tasks $f_s < 1$, we estimate the total CPU time that remains to be P_t and the longest critical path starting from any of those stages to be S_t.

4.2 Job Progress Estimation

As introduced above, we use a progress indicator to capture the state of the job and to index into $C(p, a)$, the remaining time distributions that were pre-computed from the simulator. The progress indicator should faithfully reflect the work that has happened and the work that remains. In particular, it should account for parallel stages whose tasks can finish in any order, tasks that differ widely in completion time, and stages that differ in their numbers of tasks. Furthermore, tasks sometimes fail, requiring previous output to be recomputed, and the indicator should reflect such events as well.

A job progress indicator can integrate several characteristics of a running job. Examples include the fraction of completed tasks in each stage, the aggregate CPU time spent executing, the relative time when a particular stage is started or completed, and the length of the remaining critical path. We built six progress indicators that use different subsets of these aspects. Here we describe the progress indicator that worked best in our experiments. See Section 5.4 for description and evaluation of the remaining indicators.

The *totalworkWithQ* indicator estimates job progress to be the total time that completed tasks spent enqueued or executing. Based on past run(s) of the job, we compute for each stage s, the total time tasks spend executing T_s and enqueued Q_s. At runtime, given f_s, the fraction of tasks in stage s that are complete, the progress estimate is $\sum_{\text{stage } s} f_s (Q_s + T_s)$.

This indicator is simple. In particular, it assumes that tasks in the same stage have similar queuing and running times and ignores potentially useful information such as the intra-task progress, dependencies between future stages, barriers, and the length of remaining critical path. However, our goal is to design an indicator that is an effective index into the $C(p, a)$ distributions computed in Section 4.1. Our experience (see Sections 5.2 and 5.4) shows that this indicator performs better in Jockey than more complex indicators across a wide range of conditions.

4.3 Resource Allocation Control Loop

The goal of the resource allocation control loop is to implement a policy which maximizes the job's utility and minimizes its impact on the cluster by adjusting the job's resource allocation. There are four inputs to the control loop:

1. f_s, the fraction of completed tasks in stage s
2. t_r, the time the job has spent running
3. $U(t)$, the utility of the job completing at time t. A typical utility function used in our environment would be nearly flat until the job deadline, drop to zero some time after the deadline and, in some cases, keep dropping well below zero to penalize late finishes.
4. Either the precomputed $C(p, a)$ distributions, Q_s and T_s, for each stage s (when using the simulator-based approach), or the precomputed l_s, L_s, and T_s for each stage s (when using the Amdahl's Law-based approach).

The policy's output is the resource allocation for the job.

The basic policy logic periodically observes the job's progress and adapts the allocated resources to ensure it finishes with high utility. First, it computes the progress p using a job progress indicator. Next, the expected utility from allocating a tokens is computed as follows: given progress p and the time the job has been running t_r, the expected utility is $U_a = U(t_r + C(p, a))$. Finally, the minimum allocation that maximizes utility is $A^r = \arg\min_a \{a : U_a = \max_b U_b\}$.

Inaccuracies in predicting job latencies and the non-deterministic performance of the cluster can cause the raw allocation A^r to under- or over-provision resources, or oscillate with changes. To moderate these scenarios, Jockey integrates three standard control-theory mechanisms:

1. **Slack**: To compensate for inaccuracy in the job latency estimate (by the simulator or Amdahl's Law), we multiply the predictions from $C(p, a)$ by a constant factor $\mathcal{S}$. For example, with slack $\mathcal{S} = 1.2$, we would add an additional 20% to the predictions.
2. **Hysteresis**: To smooth oscillations in the raw allocation, we use hysteresis parametrized by α. In particular, we adapt A^s_t – the smoothed allocation at time t – as follows: $A^s_t = A^s_{t-1} + \alpha(A^r - A^s_{t-1})$. Whereas a value of $\alpha = 1$ implies that the allocation immediately jumps to the desired value, for $\alpha \in (0, 1)$ the gap between the allocation and the desired value reduces exponentially with time.
3. **Dead zone**: To dampen noise in the job progress indicator, we add a dead zone of length D, i.e., shift the utility function leftwards by D and change allocations only if the job is at least D behind schedule. For example, with $D = 3$ minutes, a deadline of 60 minutes is treated as a deadline of 57 minutes, and the policy won't act unless the job is at least 3 minutes delayed.

Our results in Section 5 show the need for dynamic adaptation. We also report on the incremental contributions due to each of the above techniques. In Section 5.5 we perform a sensitivity analysis of parameter choices, and find that the slack, hysteresis, and dead zone parameters have wide operating regimes. Values for these parameters can be set in advance with the aid of Jockey's simulator: slack can be set based on simulator's margin of error when compared with actual job executions, values for hysteresis and dead zone can be determined experimentally with a simulated control loop. While the simulator does not perfectly reproduce the actual dynamics of the cluster and jobs, it provides guidance when adjusting these settings.

4.4 Limitations and Future Work

As described here, Jockey makes local decisions to ensure each job finishes within the SLO while using as few resources as necessary. We plan to extend Jockey to reach globally optimal allocations when managing multiple SLO-bound jobs. Doing so requires an additional inter-job arbiter that dynamically shifts resources from jobs with low expected marginal utility to those with high expected marginal utility.

At this time, Jockey is only capable of meeting SLOs for jobs it has seen before. We consider this a reasonable limitation since most of the business-critical jobs are recurring. For non-recurring jobs, a single profile run is enough to generate accurate job completion estimates, as demonstrated in Section 5.2. Extending Jockey to support novel jobs, either through sampling or other methods, is left for future work.

Jockey is agnostic to small changes in the input size of the job and in the execution plans. Large changes to either are visible to Jockey and can be treated as new jobs; i.e., train new completion time distributions based on the changed runs. In practice, we build Jockey's offline distributions using the largest observed input because Jockey automatically adapts the allocation based on the actual resource needs during the lifetime of the job.

We acknowledge that Jockey cannot recover from serious failures or degenerate user code. For example, if running the job on the entire cluster would not meet the SLO, Jockey is of little use. However, such cases are rare, and for common failures Jockey can meet deadlines by running the job at appropriately higher parallelism. In either case, Jockey will attempt to meet the SLO by continuously increasing the amount of resources guaranteed to the job until the model indicates that the deadline will be met, the job completes, or a hard limit is reached.

A few enhancements to the current design are also under consideration. Additional input signals to the control loop, such as the size of the input data, progress within running tasks, and cluster-wide performance metrics, could improve adaptivity. Additional control knobs such as the aggressiveness of mitigating stragglers [2], the OS priority of tasks, and the bandwidth shares of network transfers, could broaden what Jockey can do to meet SLOs. Finally, rather than use progress indicators, efficient ways to integrate the simulator with the online phase, perhaps as a less frequent control loop, could provide more precise control over job progress.

5. Evaluation

In this section, we first evaluate the ability of Jockey to meet job latency SLOs in various scenarios: different deadlines, changes in cluster conditions, and changes in deadlines during job runtime. Then, we evaluate Jockey's three components: the latency prediction, the progress indicators, and the sensitivity of the control loop to changes in its parameters in Sections 5.3, 5.4, and 5.5.

5.1 Methodology

We evaluate Jockey on 21 production jobs; these were all the recurring jobs of a business group in Microsoft that were neither too short to use in our evaluation, nor too big for the guaranteed cluster slice available to our experiments. We use a single production run of these jobs as input to the simulator (Section 4.1) to pre-compute the completion time distribution ($C(p,a)$) and other statistics (l_s, L_s, T_s, and Q_s). Experimental runs were performed using a modified job manager that implements progress indicators and adapts allocations dynamically (Sections 4.2 and 4.3). We perform more detailed analysis for a subset of these jobs; their characteristics are in Table 2 and stage dependency structure is illustrated in Fig. 3.

The analysis and experiments were performed on a large-scale cluster running production workloads with an average utilization of 80% and a number of compute nodes in the high thousands. Each node is a commodity, multi-core machine with tens of GBs of RAM. There are approximately 40 machines per rack, connected by a network oversubscribed by a small factor.

For most jobs, we evaluate Jockey's ability to meet a deadline of 60 minutes. For the detailed subset, we used two different deadlines – the longer always twice the shorter. A deadline of d minutes translates to a piecewise-linear utility function going through these points: $(0,1)$, $(d,1)$, $(d+10,-1)$, $(d+1000,-1000)$. This means that the utility drops significantly after the deadline. We ran at least three experiments for each combination of job, deadline and policy. In total, we report results from more than 800 experiments.

Our evaluation metrics are as follows: 1) did the job complete before the specified deadline?, 2) how much earlier or later did the job finish compared to the deadline?, and 3) what was the impact on the rest of the cluster for jobs that met the deadline? We measure the job's impact using the notion of *oracle allocation*. For a deadline of d minutes and a job that requires aggregate CPU time of T minutes, the oracle allocation is $O(T,d) = \lceil T/d \rceil$ tokens. This is the minimum allocation required in theory to finish the job in d minutes. This estimate is optimistic since it assumes the total work is known in advance, and the job can continuously run at a parallelism of $O(T,d)$ (that is, it is agnostic to the job's structure). However, it is a good baseline for comparing different resource allocation approaches. The job's impact on the cluster is measured as the fraction of job allocation requested by the policy that

stat	A	B	C	D	E	F	G
vertex runtime median [sec]	16.3	4.0	2.6	6.1	8.0	3.6	3.0
vertex runtime 90th percentile [sec]	61.5	54.1	5.7	25.1	130.0	17.4	7.7
vertex runtime 90th percentile [sec] (fastest stage)	4.0	3.3	1.7	1.4	3.9	3.3	1.6
vertex runtime 90th percentile [sec] (slowest stage)	126.3	116.7	21.9	72.6	320.6	110.4	68.3
total data read [GB]	222.5	114.3	151.1	268.7	195.7	285.6	155.3
number of stages	23	14	16	24	11	26	110
number of barrier stages	6	0	3	3	1	1	15
number of vertices	681	1605	5751	3897	2033	6139	8496

Table 2. Statistics of seven jobs used in evaluation.

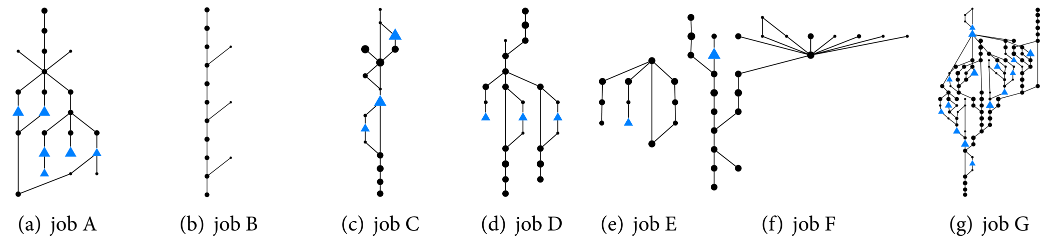


Figure 3. Stage dependencies of seven jobs used in evaluation. Each node represents a stage in a job; blue, triangular nodes are stages with full shuffle. Sizes of the nodes are proportional to the number of vertices in the stage, and edges represent stage dependencies (top to bottom). In this visualization, a typical MapReduce job would be represented by a black circle connected to a blue triangle.

is *above the oracle allocation*. See Fig. 6 for examples of the oracle allocation.

In our experiments, we re-run the resource allocation control loop (Section 4.3) each minute and use the totalworkWithQ progress indicator. We use a slack of 1.2 to accommodate the inaccuracy of job latency prediction, hysteresis parameter of 0.2 to smooth the requested resource allocation, and a dead zone of 3 minutes. We discuss Jockey's sensitivity to these values in Section 5.5, and compare progress indicators in Section 5.4.

We compare Jockey – based on predictions from the job simulator and adapting allocations at runtime – with three other policies. *Jockey w/o adaptation* uses the job simulator to find an *a priori* resource allocation that maximizes job utility, but does not adapt allocations during job runtime. *Jockey w/o simulator* does adapt but uses the simpler Amdahl's Law-based model of the job. Finally, we compare against the *max allocation* policy which guarantees all the resources available (in these experiments, 100 tokens) to finish the SLO-bound job as quickly as possible.

While the max allocation policy is able to meet all of the SLO deadlines, as shown below, it is not a practical policy to use. Because it guarantees all allocated resources to each job, it is not possible to run more than one job at a time using the max allocation policy. Because the maximum parallelism of Dryad jobs varies, running one job at a time would create "valleys" before the barrier stages, during which the resources would be underutilized (indeed, this problem would be worse for those jobs which cannot make use of all allocated resources at any point). Filling those valleys with additional

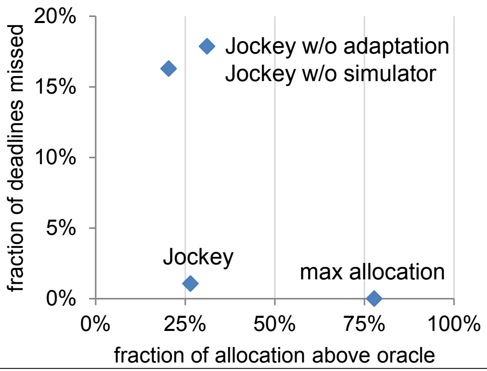
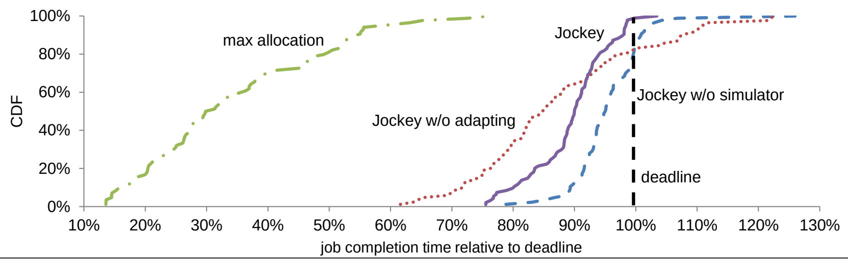


Figure 4. Comparison of average allocation above the oracle allocation and fraction of missed deadlines for each policy.

jobs naturally requires a dynamic allocation policy, as we develop with Jockey.

5.2 SLO-based Resource Allocation

Fig. 4 summarizes our experiments; there are more than 80 runs per policy. The x-axis shows the fraction of job allocation above the oracle allocation; the y-axis shows the fraction of experiments that missed deadlines. In both cases, lower values are better. Jockey misses the deadline in one experiment (see below for an explanation), and has a low impact on the rest of the cluster. Jockey w/o adaptation has a slightly higher impact on the cluster, but misses many more deadlines because of its inability to adapt to cluster changes. Jockey w/o simulator, which uses Amdahl's Law to estimate job completion times, achieves the lowest impact on the cluster, but misses many deadlines. This is because the simple analytic model of the job leads to imprecise predictions of the com-



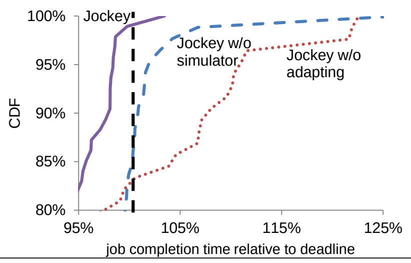


Figure 5. Left, CDFs of job completion times relative to the specified deadline for different policies. Right, detail of the upper-right corner.

pletion time. Finally, the max allocation policy meets every deadline by significantly over-provisioning the job – potentially starving other jobs in the cluster. Further, when multiple SLO-bound jobs must run simultaneously, this policy provides no benefits.

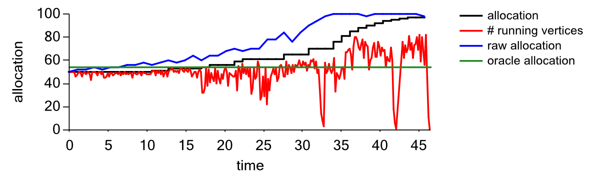
Fig. 5 presents the experimental results in more detail. The x-axis on the graph represents the job completion time relative to the specified deadline; values below 100% represent experiments that met the SLO, values above 100% (right of the dashed black line) are experiments that did not. Notice that jobs using the max allocation policy finish significantly before the deadline – the median such job finishes approximately 70% early – which translates to a large impact on the rest of the cluster; jobs under the other three policies finish much closer to the deadline. Finally, notice that using dynamic resource allocation in Jockey (solid line) further reduces the variance in latency compared to Jockey w/o adaptation (dotted line), which uses a fixed allocation of tokens.

On the right in Fig. 5, we see that while both Jockey w/o simulator and Jockey w/o adaptation miss the same fraction of deadlines, the late jobs using Jockey w/o simulator finish much earlier post-deadline. The median *late* job of Jockey w/o simulator finishes only 1% late, while the median late job of Jockey w/o adaptation finishes 10% late. This shows that even though Amdahl's Law provides less accurate latency predictions than the simulator, dynamic allocation causes jobs to finish close to the deadline. See Fig. 6 for detailed examples of experiments using our simulation-based policy.

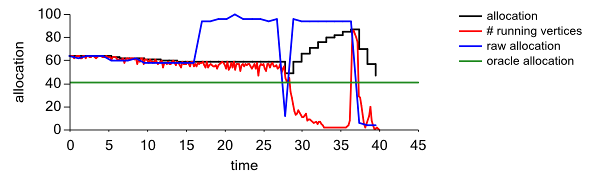
Adapting to changes in cluster conditions

Because Jockey is dynamic, it is robust to inaccurate latency predictions and can respond to changes in cluster conditions. For example, a fixed allocation calculated using a model of job performance (such as a simulator), can be too low to meet the deadline, as demonstrated above. In our experiments, the Jockey w/o adaptation policy misses the SLO deadline in 18% of experiments, even though this policy uses the same slack factor of 1.2 as our dynamic policy experiments.

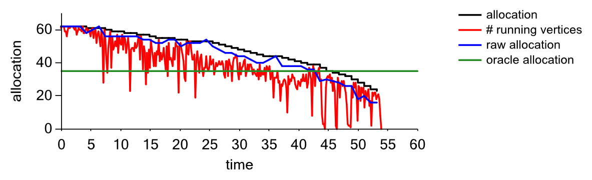
One reason such off-line predictions are inaccurate is that cluster conditions change. Because access is shared and other jobs run concurrently, use of network and CPU resources varies over time. Further detail from the experiment in which Jockey misses the SLO illustrates these variations.



(a) job F, 45-minute deadline: as described in the text, the actual job took twice as much time to execute due to an overloaded cluster. Our policy realized the slower progress and started adding resource early. In the end, the job finished only 3% late.



(b) job E, 45-minute deadline: policy started adding resources after it noticed a particular stage was taking longer to complete.



(c) job G, 60-minute deadline: policy over-provisioned the job at the beginning and released resources as the deadline approached.

Figure 6. Three examples of dynamic resource allocation policy experiments. The blue line is the raw allocation based on the job's utility and current progress, the black line is the allocation set by the policy, the red line is the number of vertices running, and the green line is the oracle allocation.

statistic	training	job 1	job 2
total work [hours]	12.7	23.5	18.5
queueing median [sec]	5.8	6.8	6.9
queueing 90th perc. [sec]	8.4	11.6	11.4
latency median [sec]	3.6	5.8	5.2
latency 90th perc. [sec]	17.4	36.6	27.1

Table 3. For job F, comparing the metrics of the training job used to create the $C(p,a)$ distributions in the simulator with two actual runs, jobs 1 and 2. Both the runs require more work; job 1 needs almost twice as much work to complete. Jockey notices the slow-down and allocates extra resources at runtime to finish job 2 on time and job 1 finishes only 90s late.

We compare the training execution that was used to compute the completion distributions, $C(p,a)$, with two actual runs of the same job when controlled by Jockey. In Table 3, *job 1* is the run that missed the deadline, whereas *job 2* met the deadline. Fig. 6(a) plots the timelapse of how Jockey adapted during job 1. Notice that the total amount of work required to finish both jobs is higher than their training runs, with job 1 needing almost twice the total work. The median and 90th percentile of vertex queueing and execution latencies are also higher. In spite of this, Jockey added enough resources to finish job 2 on time. From Fig. 6(a), we see that Jockey noticed job 1 to be slower and added resources (blue line on top), missing the deadline by only 90 seconds. Figures 6(b) and (c) show other types of adaptations. In the former, Jockey identifies a stage taking more time than usual and increases the allocation. In the latter, the job finishes faster than usual, and Jockey frees resources to be used by other jobs.

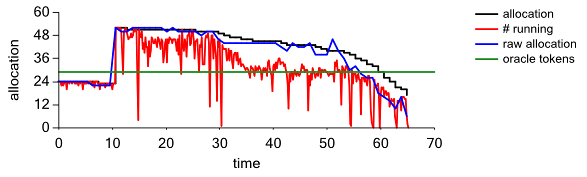
Adapting to changes in deadlines

An important feature of dynamic resource allocation is that it can adapt to changing job deadlines by adding more resources to meet a stricter deadline, or vice versa. This is crucial when multiple SLO-bound jobs must be run since we might need to slow a job to ensure a more important job finishes on time. While arbitrating among multiple SLO-bound jobs is not Jockey's focus, we view changes in deadlines as a mechanism to ensure the on-time completion of individual jobs as used by a future multi-job scheduler. The success of a such a scheduler thus depends on Jockey's ability to successfully adapt to changing deadlines. Finally, although extending a deadline does not require any change in resources in order to meet the new, longer deadline, by decreasing the amount of guaranteed resources, Jockey can make more guaranteed resources available for future SLO-bound jobs.

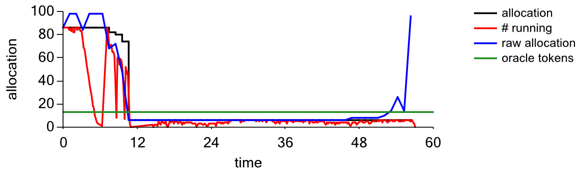
For each of the seven jobs, we performed three separate experiments in which, ten minutes after start of the job, we cut the deadline in half, doubled the deadline or tripled the deadline, respectively. In each run, Jockey met the new deadline. In the runs where we lowered the deadline by half, the policy had to increase resource allocation by 148% on average. In the runs where we doubled or tripled the deadline, the policy released 63% or 83% (respectively) of the allocated resources on average. See two example runs in Fig. 7.

5.3 Job Latency Prediction Accuracy

While our policy can adapt to small errors in latency estimates, larger errors can lead to significant over-provisioning or under-provisioning of resources. To evaluate the accuracy of the end-to-end latency predictions made by the simulator and Amdahl's Law, we executed each of the seven jobs three times at eight different allocations. We initialized the variables for both our predictors, the simulator and modified Amdahl's Law, based on jobs at one allocation and estimated their accuracy at predicting latency for other allocations. In practice, we care about the worst-case completion time, so we



(a) Deadline changed from 140 to 70 minutes. The policy adjusted the resource allocation (black line) to meet the new deadline.



(b) Deadline increased from 20 to 60 minutes. The policy released more than 90% of the resources and still met the new deadline.

Figure 7. Examples of two experiments with changing deadlines.

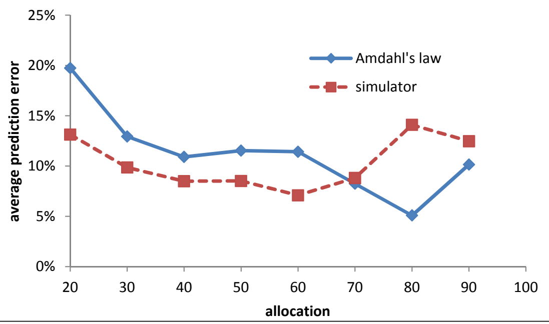


Figure 8. Average simulator and Amdahl's Law prediction error. The x-axis indicates allocations at which the job's latency was predicted.

compare the largest prediction from either predictor to the slowest run at each allocation. Across jobs and allocations, the average errors of the simulator and Amdahl's Law were 9.8% and 11.8%, respectively; see details in Fig. 8. Amdahl's Law has high error at low allocations, but performs much better at higher allocations, where the job's runtime is closer to the length of the critical path.

While the average error of Amdahl's Law is only slightly higher than the simulator's, Jockey w/o simulator missed 16% of the deadlines. The explanation of most of these SLO violations is that the policy using Amdahl's Law initially allocated too few tokens and was unable to catch-up during job run time. Also, the job simulator captures the variance of the end-to-end job latency due to outliers and failures, and therefore creates a safety buffer to make sure SLOs can be met despite this variance.

5.4 Job Progress Indicators

When dynamically allocating resources to a job, having an accurate prediction of end-to-end job latency is not enough; the control policy also needs an accurate estimate of job's progress in order to index into the remaining time distributions. Jockey uses *totalworkWithQ*, which uses the total queueing and execution time of completed vertices, to estimate job progress (Section 4.2).

Here, we describe a few other progress indicators we considered: *totalwork* computes progress as the fraction of the total execution time of completed vertices; *vertexfrac* uses the total fraction of vertices that completed; *cp* uses the fraction of the job's remaining critical path; the *minstage* and *minstage-inf* indicators use the typical start and end times of the individual stages relative to the job. If t_s^b and t_s^e are the relative start and end times of stage s, minstage infers these values from the previous run of the job while minstage-inf uses a simulation of the job with no constraint on resources and hence focusses on the critical path. Both estimate job progress as the stage furthest from when it typically completes, i.e., $\min_{\text{stage } s: f_s<1}\{t_s^b + f_s(t_s^e - t_s^b)\}$, where f_s is the fraction of vertices completed in stage s.

To evaluate these indicators, we measure how accurately they predict the end-to-end job latency during the runtime of a job. When Jockey calls the control loop at time t after the start of the job, the progress indicator estimates progress to be p_t, which is indexed into the remaining time distribution and a completion time estimate is calculated as $T_t = t + C(p_t, a)$. We compare the T_t obtained from different indicators with the actual time at which the job finishes.

The values of the progress indicator (normalized to range from 0 to 100) and the estimated completion times T_t for two of the progress indicators are shown in Fig. 9. An undesirable characteristic of a progress indicator is getting stuck (ie., reporting constant values) even when the job is making progress. We see that the CP indicator is stuck from t=20min to t=40min causing T_t to increase during this period. Such behavior confuses the control policy into assuming that the job is not making progress and increases the job's resource allocation even though the job may finish on time with the existing allocation. An ideal indicator would generate $T_t = D$ when enough resources are available, where D is the job duration, for all times t; the more T_t diverges from D, the more the control policy has to unnecessarily adjust the resource allocation of the job.

We compare these indicators using two metrics; the *longest constant interval*, i.e., the longest period, relative to the duration of the job, when the progress indicator was constant and the *average* $\triangle T$, which measures the oscillations in the T_t estimates and is computed as the average of $|T_t - T_{t+1}|$ relative to the duration of the job. The larger the value of either metric, the greater opportunity for needless oscillations in allocations. See the comparison in Table 10.

The totalworkWithQ indicator, which incorporates the duration and queueing time of vertices in each stage, performs best. The minstage, minstage-inf and CP indicators, which consider the structure of the job, perform significantly worse because their progress estimates are based on the stage which has made the least progress, and do not reflect progress in other stages. TotalworkWithQ considers progress in all running stages, and thus increments more smoothly.

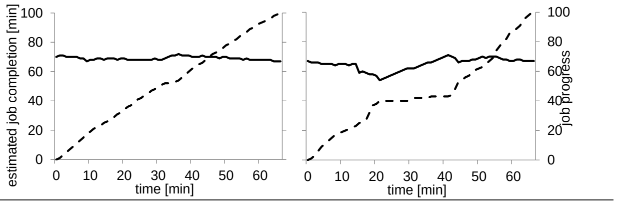


Figure 9. The totalworkWithQ (left) and CP (right) progress indicators for job G. The solid lines (left axes) show the estimated worst-case job completion times T_t, the dashed lines (right axes) correspond to the values of the progress indicator.

indicator	$\triangle T$	longest constant interval
totalworkWithQ	2.0%	8.5%
totalwork	2.3%	9.3%
vertexfrac	2.2%	10.1%
CP	3.0%	15.2%
minstage	3.3%	19.9%
minstage-inf	3.9%	26.7%

Figure 10. Comparison of progress indicators.

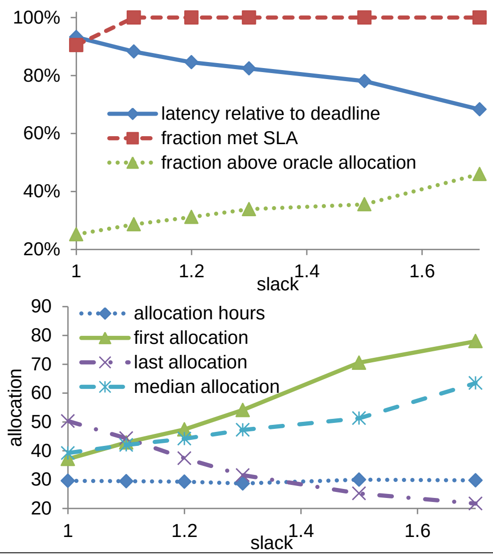


Figure 12. Sensitivity of the slack parameter. Top, the fraction of jobs that met the SLO and the fraction of job allocation above the oracle allocation. Bottom, average of first, last and median allocations during each experiment, and average of total machine hours allocated by the policy.

5.5 Sensitivity Analysis

To evaluate the control loop's sensitivity, we adjusted its parameters and ran each of the seven jobs three times with a single deadline after each adjustment. The baseline here is Jockey with default parameter values. The results are summarized in Fig. 11. Running our policy with no hysteresis and no dead zone, results in meeting only 57% of the SLOs, while using the hysteresis with no dead zone, meets 90% of the SLOs. Using hysteresis is clearly crucial; without it, the allocation fluctuates too much in each direction. When it drops too much

experiment	met SLA	latency vs. deadline	allocation above oracle	median allocation
baseline	95%	-14%	35%	52.9
no hysteresis, no deadzone	57%	-2%	25%	49.7
no deadzone	90%	-9%	30%	50.3
no slack, less hysteresis	76%	-5%	27%	44.9
5-min period	95%	-22%	35%	45.7
minstage progress	100%	-16%	34%	48.2
CP progress	95%	-16%	31%	44.9

Figure 11. Results of sensitivity analysis. The baseline results are a subset of results reported in Section 5.2.

because of this oscillation, Jockey cannot catch-up later. We tried running with no slack, but instead use an increased value of the hysteresis parameter to let Jockey adapt more quickly when jobs fall behind. Here, on average, Jockey allocated too few tokens at the start of the job and missed 24% of the deadlines. Next, changing the period at which adaptation happens from one to five minutes still met 95% of the deadlines. But, for jobs that were over-provisioned, Jockey did not quickly reduce the allocation, resulting in jobs finishing 22% before the deadlines (compared to 14% in our baseline). We also ran Jockey using the minstage and CP indicators, which met 100% and 95% of the deadlines (respectively), and had a similar impact on the cluster as the baseline. As shown in Section 5.4, these indicators have some undesirable properties as inputs to a control loop, but these experiments suggest that with hysteresis, they can still perform well in practice.

Results for different slack values are presented in Fig. 12, based on 21 runs for each value. The only SLO violations occurred in experiments without slack; adding even 10% slack was enough to meet the SLOs. Adding more slack led to jobs finishing well before the deadline and having a larger impact on the rest of the cluster because it directly causes over-allocation of resources. This can be seen in the increasing initial and median job allocations as slack is increased.

Results for different values of the hysteresis parameter are presented in Fig. 13. For each value of the parameter and each of the seven jobs, we ran three experiments. Only three experiments did not meet the SLO; two at the lower extreme value – 0.05, high smoothing – and one at the upper extreme – 1.0, no smoothing. Overall, experiments with higher values of the hysteresis parameter finished closer to the deadline and had slightly less impact on the rest of the cluster, but the maximum allocation requested by the policy was much higher than with greater smoothing.

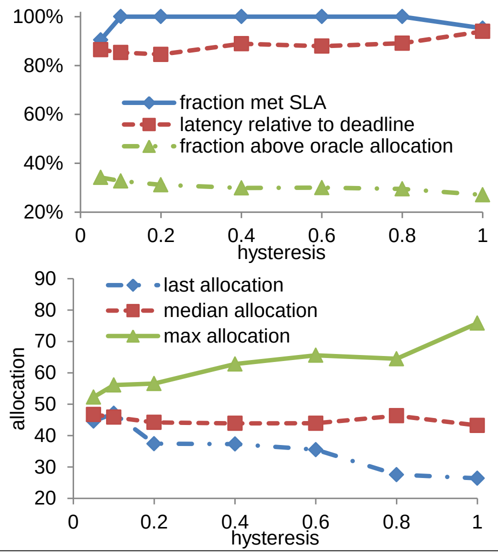


Figure 13. Sensitivity of the hysteresis parameter. Top, fraction of jobs that met the SLO, fraction of allocations above the oracle, and job latencies relative to deadlines. Bottom, average of median, max and last allocations during each run, and average of total machine hours Jockey allocated.

5.6 Summary

Our evaluation on a large-scale production cluster shows that Jockey can reliably meet job latency SLOs; in 94 experiments that ran Jockey missed one deadline by 3% due to much higher load on the cluster at that time. Without the simulator, or without dynamic resource adaptation, Jockey performed significantly worse. While the max-allocation policy met all SLOs, Jockey had 3× less impact on the rest of the cluster, which allows more SLO-bound jobs to be run simultaneously. We also demonstrated that Jockey can dynamically adapt to changing job deadlines, which allows us to trade resources between multiple jobs based on their utilities.

Jockey's success chiefly derives from an accurate model of the job, one that captures the remaining execution time as a function of resource allocation and job progress. However, when the model is not 100% accurate, standard techniques from control theory (such as hysteresis and adding slack) can partially compensate for the inaccuracy. We show in our sensitivity analysis that, without these techniques, Jockey's approach performs much worse.

Nonetheless, in certain cases, the job execution can significantly diverge from the model and the control loop could either overprovision the job (taking resources from other jobs) or underprovision it (allocating too few resources and missing the SLO). This could happen if the inputs of the job change substantially – resulting in a more expensive execution – or if the whole cluster is experiencing performance issues, such as an overloaded network. In these cases, we could quickly update the model by running the simulator at runtime, or simply fall back on weighted fair-sharing once the control loop detects large errors in model predictions.

6. Related Work

Job scheduling and resource management are not new problems, and there has been much related work in these areas. Jockey builds upon recent work on performance indicators and improvements in MapReduce-like clusters, as well as previous work in grid computing and real-time systems.

6.1 Performance in data parallel clusters

Jockey is most closely related to the Automatic Resource Inference and Allocation for MapReduce Environments project (ARIA), which also proposes a method to allocate resources in a large-scale computational framework for meeting soft deadlines [24]. Both the ARIA work and our own feature a control loop which estimates the job's progress, uses past execution profiles to predict the completion time, and adjusts the resources assigned to the task to meet the target deadline.

However, Jockey differs from ARIA in several important ways. First, the ARIA project was developed for map-reduce frameworks which feature only three computational phases: a Map phase, a Shuffle phase, and a Reduce phase; the framework used here supports directed acyclic graphs (DAGs) of arbitrarily long pipelines of independent and dependent stages. Second, the ARIA authors develop analytic equations which estimate each stage's completion time, similar to the Amdahl's Law-based approach we consider. But, as noted above, we found simulations to be more accurate for predicting future progress in DAGs because they incorporate the effects of vertex outliers, failures and barriers. Third, the approach described here is robust to changes in the cluster conditions or the deadlines or the amount of job's input. Our experiments show that slack, hysteresis and a dead zone are necessary for meeting SLOs in a production setting. Finally, the experiments here are more realistic. They are performed on a large shared production cluster with all the allied noise and a wide variety of bottlenecks. ARIA used a dedicated cluster of 66 nodes without any network bottleneck. Hence, we believe Jockey is a better match for production DAG-like frameworks such as Hive [22], Pig [17], and Ciel [16]

To predict the completion time of a running job, Jockey must estimate the job's current progress. Our approach is informed by the ParaTimer progress indicator [15], which is most similar to the vertexfrac function which we first consider. As discussed above, the vertexfrac design was not the best method for Jockey because it incorrectly indicates a lack of progress during long-running stages with a low degree of parallelism, and because it can be overly optimistic about failures and data skew [15]. When developing Jockey, we found it to be more effective to slow a job running ahead of deadline due to prior pessimism about failures, rather than attempt to speed-up a job which is running behind.

As a side-effect of predictably meeting SLOs, Jockey decreases the variance in job completion latencies. This goal is shared with work on reducing such variance directly, such as Mantri [2] and Scarlett [3]. The approach taken by Jockey, to automatically adjust resource allocations in response to predicted fluctuations in latency, is complementary to this earlier work. While we have not yet studied the effect of combining these approaches, we believe that such a combination will further improve Jockey's ability to meet SLOs.

6.2 Deadlines in grid and HPC workloads

The Amdahl's Law-like approach described in Section 4.1 is inspired by the value-maximizing, deadline-aware scheduler for animation rendering tasks by Anderson *et al.* [4], which they term the *disconnected staged scheduling problem* (DSSP). The approach developed estimates the amount of resources required to complete the task using two quantities: the aggregate CPU time (the time to complete the job on a single processor), and the length of the critical path (the time to complete the job on an infinite number of processors).

Finally, Sandholm and Lai have explored the relationship between a job's weight in a fair-sharing scheduler and the chance of meeting a given deadline in both grid-computing environments [19, 20] and MapReduce-like contexts [21]. Jockey automatically makes these weight decisions based on the utility curve submitted by the user, whereas Sandholm and Lai's methods price the system's resources based on aggregate demand and permit users to allocate resources based on their own budgets and deadlines.

6.3 Deadlines in real-time workloads

Previous work on real-time systems has advocated dynamic resource management to perform admission control [26], meet deadlines [18] and maximize aggregate utility [14], as Jockey does for data parallel clusters. Jockey differs by operating at a significantly larger scale, managing larger computational jobs with longer deadlines, and adjusting the resource allocation during a job's execution, rather than only between repeated executions of the same job. Jockey also uses a simulator to estimate a distribution of job completion times for a given allocation of resources, rather than rely upon an analytic model of the critical path.

7. Conclusion

In today's frameworks, providing guaranteed performance for pipelines of data parallel jobs is not possible on shared clusters. Jockey bridges this divide. To do so, it must combat varying availability and responsiveness of resources, two problems which are compounded by the dependency structure of data parallel jobs. By combining detailed job models with robust dynamic adaptation, Jockey guarantees job latencies without over-provisioning resources. Such "right-sizing" of allocations lets Jockey successfully run SLO-bound jobs – it met 99% of the SLOs in our experiments – in a cluster simultaneously running many other jobs – Jockey only needed 25% more resources than the theoretical minimum.

When a shared environment is underloaded, guaranteed performance brings predictability to the user experience;

when it is overloaded, utility-based resource allocation ensures jobs are completed according to importance. Jockey brings these benefits to data parallel processing in large-scale shared clusters.

Acknowledgments

The authors wish to thank Sameer Agarwal, Jonathan Perry, Moises Goldszmidt, and our shepherd Karsten Schwan for helpful comments and advice during the development of this project and while preparing our paper, as well as the many members of the Cosmos development team, particularly Jay Finger, Astha Gupta, Solom Heddaya, Pat Helland, Bikas Saha, Shoaib Sehgal, and Jingren Zhou.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS '67 (Spring)*, pages 483–485, New York, NY, USA, 1967.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. OSDI '10*, Vancouver, Canada, 2010.

[3] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proc. EuroSys '11*, Salzburg, Austria, 2011.

[4] E. Anderson, D. Beyer, K. Chaudhuri, T. Kelly, N. Salazar, C. Santos, R. Swaminathan, R. Tarjan, J. Wiener, and Y. Zhou. Value-maximizing deadline scheduling and its application to animation rendering. In *Proc. SPAA '05*, Las Vegas, Nevada, USA, 2005.

[5] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes Realtime at Facebook. In *Proc. SIGMOD '11*, Athens, Greece, 2011.

[6] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB*, 1(2):1265–1276, 2008.

[7] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Proc. PLDI '10*, Toronto, Ontario, Canada, 2010.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[10] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *Proc. SoCC '11*, Cascais, Portugal, 2011.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proc. EuroSys '07*, Lisbon, Portugal, 2007.

[12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. SOSP '09*, Big Sky, Montana, USA, 2009.

[13] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proc. IMC '10*, Melbourne, Australia, 2010.

[14] C. Lumezanu, S. Bhola, and M. Astley. Online optimization for latency assignment in distributed real-time systems. In *Proc. ICDCS '08*, June 2008.

[15] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *Proc. SIGMOD '10*, Indianapolis, IN, USA, 2010.

[16] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. NSDI'11*, Boston, MA, 2011.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. SIGMOD '08*, Vancouver, Canada, 2008.

[18] B. Ravindran, P. Kachroo, and T. Hegazy. Adaptive resource management in asynchronous real-time distributed systems using feedback control functions. In *Proc. of 5th Symposium on Autonomous Decentralized Systems*, 2001.

[19] T. Sandholm and K. Lai. Prediction-based enforcement of performance contracts. In *Proc. GECON'07*, Rennes, France, 2007.

[20] T. Sandholm and K. Lai. A statistical approach to risk mitigation in computational markets. In *Proc. HPDC '07*, Monterey, CA, USA, 2007.

[21] T. Sandholm and K. Lai. MapReduce optimization using regulated dynamic prioritization. In *Proc. SIGMETRICS '09*, Seattle, WA, USA, 2009.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2: 1626–1629, August 2009.

[23] B. Urgaonkar, P. Shenoy, A. Ch, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Proc. ICAC '05*, 2005.

[24] A. Verma, L. Cherkasova, and R. H. Campbell. SLO-Driven Right-Sizing and Resource Provisioning of MapReduce Jobs. In *Proc. LADIS '11*, 2011.

[25] G. Wang and T. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proc. IEEE INFOCOM '10*, March 2010.

[26] D. Xuan, R. Bettati, J. Chen, W. Zhao, and C. Li. Utilization-Based Admission Control for Real-Time Applications. In *Proc. ICPP '10*, Washington, DC, USA, 2000.

[27] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. EuroSys '10*, Paris, France, 2010.