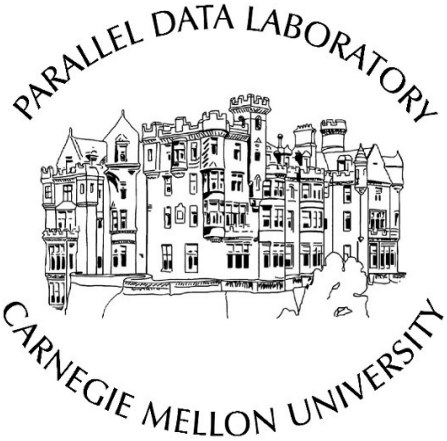


JamaisVu: Robust Scheduling with Auto-Estimated Job Runtimes

Alexey Tumanov*, Angela Jiang*, Jun Woo Park*
Michael A. Kozuch†, Gregory R. Ganger*
Carnegie Mellon University*, Intel Labs†

CMU-PDL-16-104
September 2016

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

JamaisVu is a new end-to-end cluster scheduling system that automatically generates and robustly exploits job runtime predictions. Using runtime knowledge allows it to more effectively pack jobs with diverse time concerns (e.g., deadline vs. latency) and soft placement constraints on heterogeneous cluster resources. JamaisVu's job run time predictor, JVuPredict uses a new black-box approach that tracks job run time history as a function of multiple job submission features (e.g., user ID and program name), and then adaptively uses the most effective feature subset for each submitted job. Analysis of a 1-month Google cluster trace shows JVuPredict predicts reasonably well for complex real-world job mixes; for example, 90% of predictions are within a factor of two of actual runtime. But, because predictions cannot be perfect, JamaisVu includes new techniques for mitigating the effects of such real misprediction profiles. Experiments with workloads derived from the trace show that JamaisVu performs nearly as well as a hypothetical scheduler with perfect job runtime information, outperforming runtime-unaware scheduling by reducing SLO miss rate, increasing goodput, and maintaining comparable latency for best effort jobs.

Acknowledgements: We thank the member companies of the PDL Consortium (Broadcom, Citadel, Dell EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate Technology, Tintri, Two Sigma, Uber, Veritas, Western Digital) for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by an NSERC Postgraduate Fellowship, by a Samsung Scholarship, and by National Science Foundation under awards CSR-1116282, 0946825 and CNS-1042537, CNS-1042543 (PRObE).

Keywords: cluster scheduling, cloud systems

1 Introduction

Modern cluster schedulers face a daunting task. Modern clusters are used for a diverse mix of activities, including exploratory analytics, application development and test, scheduled content generation, and customer-facing services [21]. Pending work should be mapped to the heterogeneous resources so as to satisfy deadlines for business-critical jobs, minimize delays for interactive best-effort jobs, maximize efficiency, and other goals. Cluster schedulers are expected to make that happen.

Knowledge of job runtimes has been identified as a powerful building block for modern cluster schedulers [28]. Runtime knowledge allows a scheduler to pack jobs more aggressively in a cluster's resource assignment plan [6, 28, 32], such as by allowing a latency-sensitive best-effort job to run before a high-priority batch job if it will still meet its deadline. Runtime knowledge allows a scheduler to determine whether it is better to start a job immediately on suboptimal machine types with worse expected performance, or wait for the jobs currently occupying the preferred machines to finish (or to preempt them) [28, 2]. Exploiting job runtime knowledge leads to better, more robust scheduler decisions than relying on hard-coded assumptions.

A natural set of questions, then, ask where runtime knowledge comes from, how accurate it is, and what happens when it is imperfect. Previous work has offered a few knowledge sources, including the dubious possibility that users could provide runtimes and better options for two particular classes of jobs. For jobs with a known structure, such as Dryad [12, 13] or map-reduce jobs, performance models can be parameterized with profiling (historical or once started) to produce good predictions [9, 1, 33, 18]. For the simple case of known-to-be-repeating jobs, such as with periodic submissions of a given analytics job, simple history-based predictions work well [6]. But, these available options leave the questions unresolved for the many jobs in consolidated clusters that do not declare a known structure or history.

This paper describes JamaisVu, a cluster scheduling system that automatically generates and robustly exploits runtime predictions for all submitted jobs. JamaisVu retains histories of job runtimes and uses them to predict new jobs' runtime, when it is submitted. Many jobs, while never-before-seen (FR: *jamais vu*), are similar to previous jobs, even when not declared repeats. Examples include a developer's regular test routines, a designer's scripted simulator parameter sweep, or an analyst's training of different ML models in search for a better one. The key is to identify the right previous jobs to focus on, for a given new job. To do so, JamaisVu's prediction module JVuPredict uses features (e.g., user, program name) of submitted jobs to associate a new job with particular job history. Importantly, it adaptively determines which features to rely on for a given job and how to aggregate selected history (e.g., how heavily to weight recency), because no single feature:estimator pair is best for all cases.

Evaluation of prediction accuracies for the jobs in the well-studied Google cluster trace [21] show that JamaisVu produces good predictions for most jobs. For example, ≈90% of predictions are within 2X of the actual runtime, and most are much closer. But, few are perfect. As might be expected, most runtimes are over- or under-predicted by moderate amounts, some by 50–100%, and a few by an order of magnitude or more (e.g., an unexpected corner case for a given program). So, the core scheduler must be able to cope with a variety of mispredictions among the mostly good runtime predictions.

The scheduler core, called JVuSched, follows the increasingly popular approach of packing cluster space-time via an optimization algorithm [6, 28, 32]. It builds on a state-of-the-art scheduler, called TetriSched [28], which exploits job runtime knowledge in using a mixed integer linear program (MILP) to create the plan and regularly revises the plan to incorporate new jobs and deal with minor mispredictions. But, we found that it was easily mislead into quite poor behavior by the breadth of mispredictions arising from black-box predictions of diverse real-world jobs.

JVuSched integrates three mechanisms to mitigate problems caused by such mispredictions.

1. To address under-estimates, discovered when a job continues longer than expected, careful increasing of the expected runtime is needed. Increasing too slowly can cause the scheduler to let it keep running longer than it should because it keeps expecting the job to finish very soon. Increasing too quickly

can cause the scheduler to give up on it too soon (recall that most estimates are relatively accurate). JVuSched uses a slow-starting exponential increase.

2. To address over-estimates, especially for jobs with deadlines, a positive but diminishing value is associated with completing the job soon after the deadline. Without this, an over-estimated job can be viewed as impossible to complete in time (even when may not be), so the optimizer would appropriately choose not to waste resources on it.

3. To rigorously determine when running jobs should be stopped in favor of pending jobs, JVuSched integrates the first two mechanisms into and extends the MILP problem to explicitly consider each job's preemption cost/benefit.

Overall, real system and simulation experiments with production-derived workloads demonstrate JamaisVu's effectiveness. Using imperfect but automatically-generated runtime predictions, JamaisVu greatly outperforms runtime-unaware scheduling, especially when scheduling a mix of deadline-oriented jobs and latency-sensitive best-effort jobs on heterogeneous resources. JamaisVu provides higher SLO attainment for the deadline-oriented jobs, higher goodput, and lower latency for best-effort jobs. In most cases, JamaisVu performs nearly as well as a hypothetical system with perfect runtime information.

This paper makes four primary contributions. First, it describes the first end-to-end scheduling system based on aggressive use of automatically generated black-box runtime predictions. Second, it describes a new black-box runtime predictor that adaptively selects, for each job, the feature-history:estimator pair that should predict it best. It shows that this new predictor, JVuPredict is effective at predicting runtimes for the diverse workloads reflected in the Google cluster trace, and characterizes the mispredictions. Third, it describes new core scheduler mechanisms, implemented in JVuSched, needed to mitigate the effects of significant over- and under-predictions, including large outliers. Fourth, it reports on end-to-end experiments on a real 256-node cluster, showing that the resulting end-to-end scheduling system robustly exploits its runtime predictions to improve SLO attainment and best-effort performance, dealing gracefully with mispredictions. Indeed, JamaisVu often performs almost as well as a hypothetical scheduler with oracular runtime knowledge.

2 Background and Related Work

Cluster consolidation in modern datacenters forces cluster schedulers to handle a diverse mix of workload types, resource capabilities, and user concerns [21, 32, 22]. The result has been a resurgence in cluster scheduling research and new approaches to increasing the effectiveness of large multi-purpose clusters. This section focuses on work related to the use of one particular building block—job runtime predictions—to make better scheduling decisions

2.1 Exploiting job runtime predictions

Recent scheduling work [28, 9] shows that accurate job runtime predictions can be exploited to significant benefit when dealing with workload and resource diversity. This subsection (2.1) discusses three ways of exploiting the submission-time runtime predictions on which we focus, as well as another form of runtime prediction exploitation (elastic job sizing) that is outside this paper's scope.

Resource type assignment in heterogeneous clusters. Datacenter resources, today, are increasingly heterogeneous; consolidation of multiple workloads onto a shared infrastructure does not remove the efficiency benefits of resource specialization. Rather than having separate clusters for each machine type, modern clusters consist of a mixture of machines with different sizes (e.g., huge-memory machines), special accelerators (e.g., GPUs or FPGAs), and ages (due to incremental cluster growth and refresh). Various asymmetries may also arise, such as in interconnect topology (e.g., per-rack switches) and data locality (e.g., cached executables and/or input data).

Jobs will be affected differently by specific machine assignments. Some jobs may execute largely the same on any machines, a common assumption in historical schedulers; for others, only certain assignments

may be acceptable. A growing category of jobs, however, are those with *soft constraints* [26, 28, 27]; such jobs can accept many assignments but will run faster or more robustly given specific assignments (e.g., on a GPU-equipped machine or with all tasks on the same rack). Maximizing cluster effectiveness in the presence of jobs with such *soft constraints* requires careful scheduling [26, 28].

Scheduling jobs with soft constraints is more effective when job runtimes are known [28, 2, 36]. Naturally, the scheduler may assign the *preferred* resources, provided that they are available when a new job is first considered for scheduling. But, if they are not, the scheduler may need to decide whether assigning less-preferred resources immediately is better than waiting for preferred resources to become available (and/or freeing them via preemption).

In the absence of runtime knowledge, some schedulers [29, 36] use *delay scheduling*—waiting for a small, pre-configured amount of time for preferred resources to become available and falling back to alternatives if they remain occupied. Of course, sometimes jobs wait when they should not, and sometimes they don't wait long enough.

Packing deadline-oriented jobs with latency-sensitive jobs. Cluster workloads are increasingly a mixture of business-critical production jobs and best-effort engineering/analysis jobs. The production jobs are often workflows submitted periodically by automated systems [14, 24] that consume significant resources (e.g., many servers and tens of TBs of data), run for hours, and have strict completion deadlines(i.e., completion-time Service Level Objectives, or SLOs). The best-effort jobs, such as exploratory data analytics and software development/debugging, while lower priority, are often latency-sensitive. Schedulers can more effectively order jobs when given their runtimes, simultaneously increasing SLO attainment for production jobs and reducing average latency for best-effort jobs [6, 28].

Gang scheduling with back-filling. A common requirement of high-performance computing (HPC) jobs is that all tasks making up a job be initiated and executed simultaneously [20, 19]. Such *gang scheduling* requires a sufficient set of machines to be available to assign at once. To avoid starving large jobs, HPC schedulers often reserve machines until the required number become free. Since leaving machines idle is undesirable, HPC schedulers may employ *backfilling* [17, 8, 25, 37] to opportunistically execute small jobs on the otherwise-idle compute capacity that will be used for the large job. Gang-scheduling and backfilling are most effective when job runtimes are known.

Elastic sizing of jobs based on progress. Jobs that expose progress metrics and are elastically parallelizable can be dynamically resized to achieve completion-time targets [9]. We differentiate this use of runtime predictions from our focus on those that are available when deciding when and where to start a job—that is, available before the job starts running.

2.2 Predicting job runtimes

In some environments, especially HPC and grid computing environments, users have been expected to provide runtime information explicitly. Naturally, the quality of such user-provided information varies widely, and automated approaches to generating runtime predictions is desirable.

Strategies for predicting job execution times may be categorized according to how much is assumed or known about a job. First, some techniques [31, 16, 6, 15, 7] are designed for explicitly repeating jobs, such as in a scripted simulation parameter sweep or regular post-processing of an output file. So, each such job is a *recurrence* of a nearly identical job with known historical runtime information.

Second, performance modeling based on *white-box techniques* assumes that the structure of each job is known. This information, together with input file characteristics, feeds performance models used for runtime prediction. For example, Jockey [9] and Perforator [1] leverage job structure and combine it with profiling for accurate runtime predictions. MapReduce's map-shuffle-reduce structure is well-understood and lends itself to analytical performance models, such as ARIA [33] and Parallax [18]. Similarly, Apollo [2] and Ernest [30] rely on leveraging job structure knowledge to estimate job runtimes.

A third category of runtime predictors, into which JamaisVu's predictor fits, uses *black-box techniques* to

address the many jobs that neither (a) report explicit recurrence nor (b) arrive with known white-box models. Few such predictors have been reported in the literature. Harchol-Balter and Downey [11] showed that, in the absence of other information, a reasonable approach to predicting a job's *remaining* runtime is to assume it is half-way completed. The closest prior work categorized HPC jobs, such as by user or by site, during post-processing of Grid traces, to characterize and determine the predictability of job runtimes and queue wait times [23]. JamaisVu's predictor (Sec. 3) goes beyond prior approaches by applying online learning techniques to identify adaptively the best job characteristics by which to categorize and the best predictor to use for each resulting category. JamaisVu generates sufficiently accurate predictions even for the extremely diverse collection of jobs submitted to a Google production cluster [21].

2.3 Addressing runtime mis-predictions

Of course, no runtime prediction mechanism will be perfect in practice. Mis-predicted runtimes can lead to sub-optimal scheduling. For example, an under-estimated runtime may result in a job starting too late to finish by its deadline. Job runtime over-estimates may result in other jobs being run less efficiently on non-preferred resources rather than waiting for preferred (but expected to be occupied) resources. Often, when a packing algorithm makes decisions based on inaccurate job "shapes", cluster efficiency decreases. Yet, we are aware of no work that analyzes and addresses specific effects of mispredictions on cluster scheduling.

Preemption is one standard tool that can be applied to address some issues arising from mispredictions, either by killing (e.g., in container-based clusters [34]) or migrating (e.g., in VM-based systems [35]) jobs. Traditionally, preemption is used to re-assign resources occupied by lower-priority jobs to higher-priority jobs. However, it can also be used to address situations where under-predicted runtimes result in a job unable to complete within the time-window planned for it or to give preferred resources to jobs that would benefit more– thereby raising overall cluster efficiency. We introduce a new approach to making preemption decisions (Sec. 4.1) that regularly reviews the schedule holistically and determines the overall cost and benefit of potential preemption-scheduling combinations.

Preemption alone is insufficient, and JamaisVu introduces several refinements (Sec. 4.2 and 4.3) specifically aimed at mitigating the negative effects of mispredictions. For example, when a job exceeds its (under-)predicted runtime, careful online re-prediction is needed to make good decisions regarding preemption. For jobs with short deadlines, some hedging is needed in the benefit model to increase the likelihood that the scheduler will dispatch those jobs despite long predicted runtimes– because those runtimes may be over-estimated. Our experiments show that these refinements are critical to achieving robust runtime-aware scheduling in the face of inevitable mispredictions.

3 JVuPredict: JamaisVu's runtime predictor

JVuPredict uses a black-box approach to predict the runtime of each job, at job submission time. Our design does not require knowledge of job structures, user-provided runtime estimates, or explicit declarations of similarity to specific previous jobs. However, we do assume that, even in multi-purpose clusters used for a diverse array of activities, most jobs will be similar to some previous jobs. For example, all runs of a particular program may be similar, as might the job submitted each night by the "user" that is responsible for daily report generation. Under this assumption, given a history of executed jobs, the runtime of a newly-submitted job may be predicted– provided that the right set of correlated previous jobs may be identified.

JVuPredict uses information (e.g., user, program name, resources requested) already specified for submitted jobs to associate a new job with particular previous jobs, adaptively determining which information to rely on and how to weight recency. This section describes JVuPredict in more detail, including information considered and estimators used. We use a trace [21] containing a month of job submission and execution data from a 12,000-node cluster at Google to motivate aspects of JVuPredict's design as well as to evaluate it (in Section 6.2).

3.1 Overview

Fig 1 illustrates JVuPredict's runtime prediction workflow. Each job is submitted with a set of *features*. The example has two features, user ID and program name, but other common features include submission time, priority level, and resources requested (containers, cores, RAM). JVuPredict tracks and uses per-feature-value job runtime history for each of multiple features because no single feature is sufficiently predictive for all jobs.

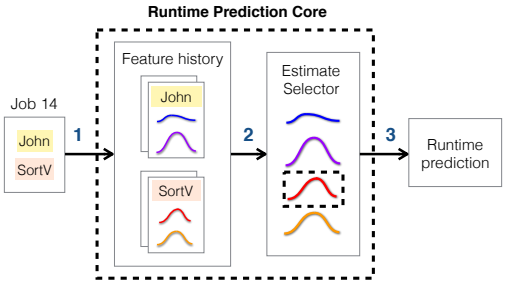


Figure 1: JVuPredict runtime prediction workflow.

For each feature being tracked (Section 3.2), the runtime history matching the new job's value for that feature is looked up (step #1). Then, from each such history, runtime predictions derived from multiple estimators (Section 3.3) are retreived (step #2); an assessment of the past performace for each feature-value:estimator pair is also retreived. The pair with the best past performance is then chosen as the final prediction that will be used in scheduling (step #3). This workflow allows JVuPredict to adaptively identify the best option for predicting particular jobs' runtimes.

Not shown in the illustration is the history collection process. As each job completes, its actual runtime is recorded and incorporated into each of the appropriate feature-value histories to generate new prediction values. Also, the real runtime is compared to the previous prediction produced by each relevant feature-value:estimator pair, and each associated performance metric is updated.

3.2 Choosing features to track

One can evaluate the potential of a feature by looking at its ability to predict accurately, as well as its coverage of jobs. These metrics are often conflicting. Features that aggressively differentiate jobs may have less variance, leading to more accurate estimates. However, the history available for these distinctive features may often be insufficient for good predictions.

Feature	Description	% Prediction error over 2X	% Unique Occurrences
Job Name (job)	User provided job name	24%	38.25%
Logical Job Name (ljob)	Application name (e.g. MapReduce)	12%	8.37%
User ID (user)	Google engineers and services name	33%	0.04%

Table 1: JVuPredict-tracked features for Google cluster trace. "% Prediction error over 2X" indicates the percentage of really bad predictions: under-predicted by 50% or over-predicted by 100%. "% Unique Occurrences" indicates the percentage of jobs for which the feature had a value not seen earlier in the trace, meaning JVuPredict would have no history for it.

The features that we use from the Google cluster trace can be found Table 1, which also coarsely summarizes the predictability and coverage of each feature. Among 385,582 jobs in the trace, 38.25% of submissions use a job name that does not appear earlier in the trace. Meanwhile, user ID provides nearly 100% coverage of jobs, but is less tightly correlated with runtimes. Fig. 2 illustrates why, showing job

runtimes for two users, one of which (User 35) has a very steady profile but the other of which (User 27) exhibits a bimodal distribution that might result from submitting two distinct types of job.

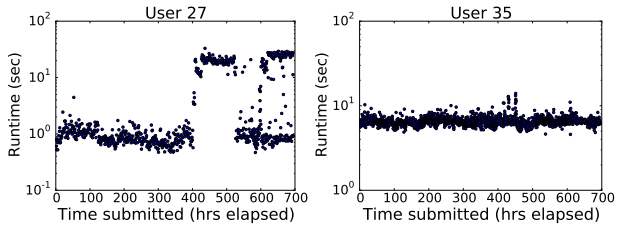


Figure 2: Runtimes of jobs submitted by each of two users.

To evaluate the predictability of a feature, we analyze the Google cluster trace and make runtime estimates from the feature's job history. The trace is replayed through JVuPredict, with the traced runtimes being reported to it as the actual runtimes. Fig. 3 shows the accuracy of estimates generated using different feature histories, to illustrate their individual effectiveness. (The accuracy of JVuPredict's overall accuracy is evaluated in Section 6.2.) The X-axis shows prediction accuracy as a percentage from actual runtime, with negative values corresponding to under-estimates and positive values to over-estimates. The X-axis goes from -100% to +100%, with the spikes at +100% indicating all values with larger over-estimates. When there is no history for a feature, a near-zero underestimate is provided, leading to a relative error of -100%. Each line corresponds to the prediction accuracies that would be seen if JVuPredict were to always use the particular feature:estimator pair for prediction. Jobname is an example of a feature that is capable of predicting well, but is too often unique or almost unique. While Jobname provides the most estimates within 5% of the actual runtime, other features provide more coverage and are more widely applicable. The two flattest lines (all-rolling and all-medium) correspond to treating all jobs as one group, showing that grouping by feature is critical to producing effective predictions.

One may not expect features such as the number of cores requested, memory requested, and number of tasks to be predictive alone. However, these "secondary" features can further refine the clusters made by our primary features (e.g. job name, user). We perform a Kruskal-Wallis (KW) test on clusters that are created using our primary features and then further distilled based upon the total cpu and memory requested for a job. Among users' histories, which are the most variable of the primary features, the most discerning clusters come from taking into account total cpu requested on groups with over 20 samples. The KW test gives a p-value of 0.078, implying a 92% confidence that the refined clusters come from different distributions. However, like other features, resources requested will not always provide better estimates. Therefore, we provide JVuPredict the option to use or to ignore secondary features. For instance, JVuPredict makes estimates based on the job name, as well as the job name and resources requested. We also discretize our secondary features, to ensure that clusters have enough samples to make estimates from. For secondary features between 0 and 1, we round the value to the nearest tenth. For values greater than one, we round to the nearest integer. Sec. 6.2 evaluates the benefit of using secondary features.

3.3 Estimators and selecting predictions

With diverse workloads, different estimation techniques will be more predictive for different jobs. For example, for feature-values with heavy-tailed runtime history distributions, median may be a better predictor than an average that weights more recent values more heavily. Internally, JVuPredict continuously makes multiple predictions and determines online which feature-value:estimator pairs are predictive on a job-by-job basis. Specifically, JVuPredict uses four estimation techniques with each feature-value:

Average Average over entire history for a feature-value
Median Median over entire history for a feature-value

Rolling Exponentially weighted decay of history, with $\alpha = 0.5$
Recent Average of the most recent 20 runtimes

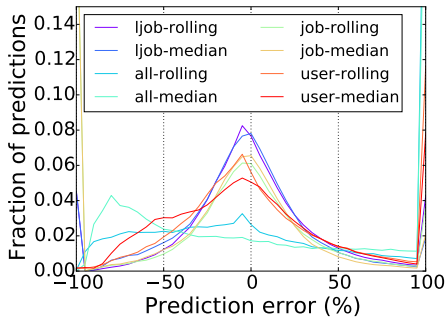


Figure 3: Distribution of relative estimate error percentages of eight feature:estimator pairs.

For a given job, JVuPredict will produce a prediction for every combination of feature and estimation policy and then choose one of these candidate predictions for use in the scheduler core by evaluating online the predictiveness of each estimator. To do so, it tracks the root mean squared error (RMSE) of past predictions, normalized by the average runtime, for each feature-value:estimator pair. The candidate prediction from the pair with the lowest RMSE is selected. We also evaluated the effectiveness of using unnormalized mean squared error and root mean squared error normalized by the variance of runtimes, but both performed less well.

JVuPredict continuously (and incrementally) updates the feature-value:estimator RMSE values, as each job completes and thereby exposes its actual runtime and each candidate prediction's error. So, the feature-value:estimator pairs that perform best are determined automatically and used by JVuPredict. This approach is nicely extensible, making it easy to change the set of features tracked and the set of estimators used, which has been convenient for experimentation and would be valuable in practice as well.

3.4 Outlier detection

In our analysis, we find that 1% of JVuPredict's estimates are considered outliers, where an outlier is defined to be a runtime over three standard deviations away from the mean of a feature history. While the number of outliers is manageable, their impact on future estimates can be large. Over 57% of feature histories contain at least one outlier. In order to mitigate these effects, we apply a median filter to runtime clusters, providing another set of options for JV to choose from. An example of the effects of a median filter are shown in Fig. 4.

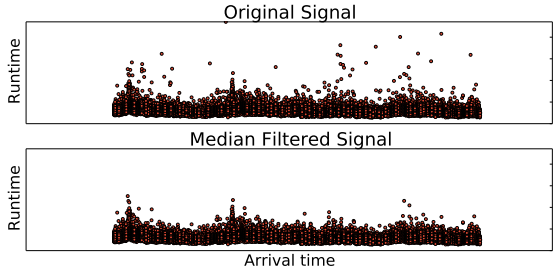


Figure 4: Effects of a median filter on runtimes from an example logical job name.

4 JVuSched: JamaisVu's scheduler

This section describes the design and implementation of JVuSched, which relies on runtime predictions to make better decisions (as many recent schedulers do), but also includes novel support for "in situ" preemption—ability to make preemption decisions simultaneously with resource allocation decisions. This enables JVuSched to make better cost/benefit tradeoffs involving preemption and improve misprediction handling.

At it's core, JVuSched uses an MILP solver to consider all pending and running jobs each scheduling cycle. Each pending job is internally encoded by an algebraic expression that describes options for executing it (which resources for how long) and an associated value. Each running job also has an expression describing the cost of preempting all or part of its current resource assignment. Each cycle, these expressions are merged and converted into a single MILP formulation processed by an off-the-shelf solver for a bounded amount of time, allowing the costs and benefits of different outcomes to be considered holistically and simultaneously. The result is extracted and interpreted to determine which running jobs get killed and which pending jobs are initiated on which resources. These decisions are then communicated to the resource management framework—YARN in our implementation—which is responsible for job and resource life-cycle management.

JVuSched is implemented by modifying an existing YARN-integrated scheduling system called TetriSched [28, 27]. As such, JVuSched benefits from TetriSched's support for soft constraints on heterogeneous cluster resources (e.g., machine attributes, rack locality, failure domains), MILP-based scheduling, and automatic generation of basic algebraic expressions for SLO and best-effort jobs. This section describes JVuSched's novel support for preemption decision making and for explicitly addressing runtime under- and over-estimates.

4.1 In Situ Preemption

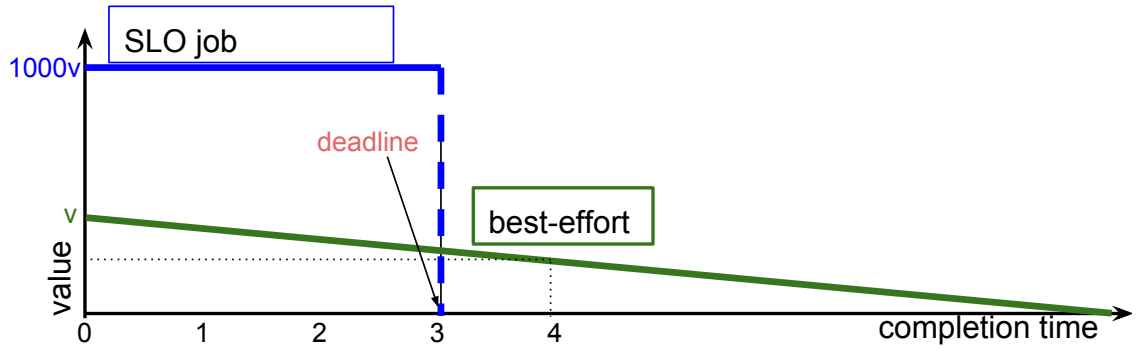
4.1.1 JVuSched job description language

JVuSched uses an internal language derived from TetriSched's space-time request language (STRL). A submitted STRL expression encodes a job's resource request in a concise algebraic language that describes resource type and topology options and their relative value as well as temporal considerations like deadlines and latency sensitivity. JVuSched needs more, requiring an extended language that can also describe the cost associated with preempting all or part of the resources currently being used by a running job.

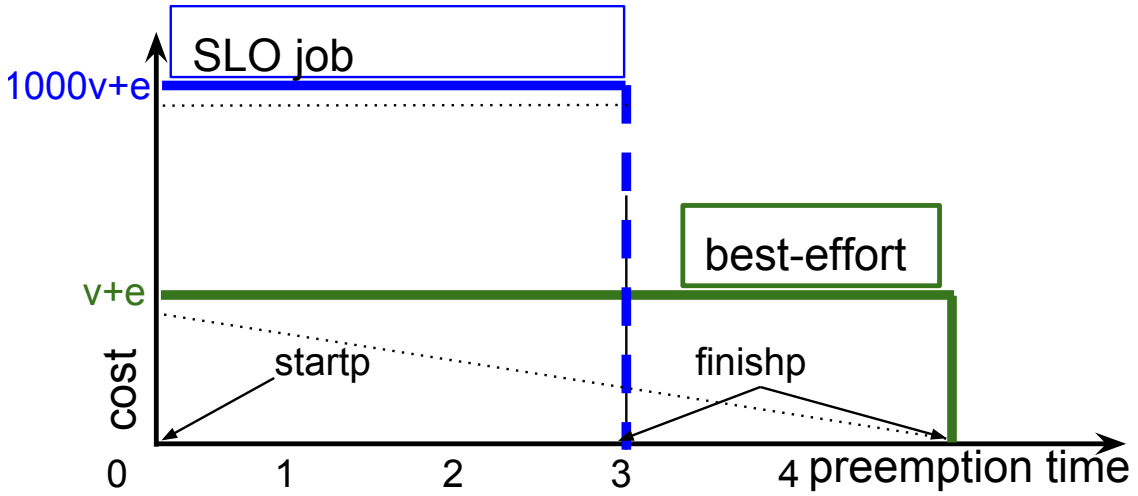
The principal STRL language primitive, nCk, associates value v with arbitrary rectangles in cluster resource space-time: *nCk*(equivalence_set, k, $start$, dur, v). It leverages grouping resources identical from a job's perspective into equivalence sets—nCk's first parameter. Then, nCk specifies the quantity desired from this equivalence set and delineates the start and expected finish time when these resources will be used. Lastly, it associates a scalar value with this space-time rectangle. Composing nCk primitives algebraically with a handful of operators (most notably sum, min, max) makes it possible to express any arbitrary resource space-time shape to the scheduler succinctly.

For JVuSched, we also need a language primitive that enables us to capture the cost associated with preempting a currently used space-time allocation. We introduce a new preemption primitive, KillnCk, that captures two-dimensional space-time preemption options. Its generality enables jobs to declaratively specify equivalence sets of resources and the quantity that can be preempted from that set. For example, suppose that a job currently holds GPU and non-GPU resources, where GPU nodes are used for tightly coupled non-elastic compute-intensive activity. The job can specify that preempting anything from the GPU equivalence set is very expensive, while preempting non-GPU resources is a lot cheaper. Thus, KillnCk can encode $O(\binom{n}{k})$ possible k-sized preemption options from an equivalence set of n nodes with as few as 1-2 KillnCk primitives. Formally, our pSTRL extensions include:

1. A "kill n Choose k" expression of the form:
KillnCk(equivalence_set, k, $startp$, $finishp$, c). It is the main pSTRL primitive used to represent a choice of any k resources out of the specified equivalence set for preemption, where $startp$ marks the earliest possible



(a) value functions



(b) preemption cost functions

Figure 5: Internal JVuSched preemption cost functions for SLO and BE jobs.

time when preemption is allowed, and $finishp$ marks the last opportunity to preempt this job – typically its expected completion time. A linear "kill n Choose k" is a variant useful for encoding willingness to preempt up to but not exceeding k nodes from an n-sized equivalence set of nodes. The cost of preemption c is sized linearly with the number of nodes chosen by the solver to be preempted.

2. a MAX expression of the form $max(e_1, ..., e_n)$ returns the value of its highest valued subexpression. MAX carries the semantics of exclusive OR (when the overall expression is maximized). It is useful to represent mutually exclusive preemption options (e.g., either 8 nodes from rack 1 or 16 nodes from rack 2, but not across racks).

3. a SUM expression of the form $sum(e_1, e_2..., e_n)$ returns the sum of its subexpressions' values. The SUM operator can be used to aggregate all possible preemption options and carries the semantics of their union (e.g., preempt 8 nodes on rack1, 16 nodes on rack2, or both).

4.1.2 Preemption cost functions

We must inform the scheduler about the cost associated with preempting a given job. We model the cost of preemption as a function of time $c(t)$ (Fig. 5). For a given job, $c(t)$ is defined only in the range specified by the KillnCk interval $[startp; finishp]$. The job can only be preempted within this interval of time. SLO jobs are modeled to have the highest cost of preemption, while best-effort jobs—the lowest. **The initial value of $c(t)$:** is empirically chosen to impose strict initial priority among jobs. For example, the initial value for SLO jobs admitted by the admission control system is set to $1000\times$ the initial value of best effort jobs. We add a small ε to ensure that jobs already running cannot be preempted by equally valued pending jobs.

```
gen: (expr , indicator var) → objective function
func gen(expr, I):
    switch expr :
        case KillnCk(partitions, k, startp, finishp, c)
            foreach x in partitions :
                P_x := integer variable  // Create partition variable
                for t := startp to finishp :
                    // (Supply) Track usage
                    Add −1 ∗ P_x to used(x, t)
            // (Demand) Ensure this node
            // gets k servers if chosen
            Add constraint Σ_x P_x = k ∗ I
            return −1 ∗ c ∗ I  // Return value if chosen
// Main function
I := binary variable  // Create indicator variable
f = gen(expr, I)
foreach x in partitions :
    for t := now to now + horizon :
        // (Supply) Ensure usage ≤ avail resources
        Add constraint Σ_{P∈used(x,t)} P ≤ avail(x, t)
solve(f, constraints)
```

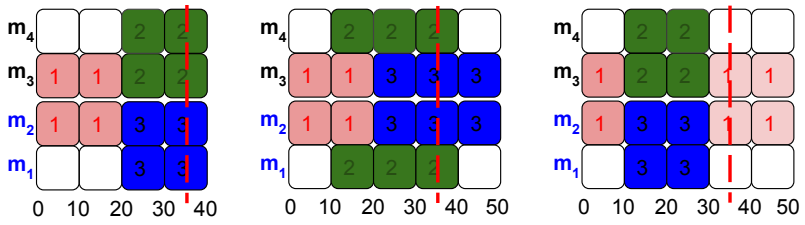
Algorithm 1: MILP generation: preemption primitive

JVuSched MILP generation. Each scheduling cycle, JVuSched aggregates all pending job requests into a single expression, automatically compiles it to a MILP, and solves it to determine any new resource assignments or revocations needed. This subsection outlines the MILP generation process for supporting simultaneous consideration of assignment and preemption.

A canonical MILP instance consists of three main components: decision variables—the unknowns the solver attempts to provide an answer for, constraints—linear inequalities based on decision variables that must be satisfied by the solver, and an objective function—a linear function on decision variables optimized by the solver. The tree structure of STRL algebraic expressions lends itself nicely to a recursive MILP generation algorithm. Starting from the root of the expression, we gradually (a) create new decision variables as needed for each node of the tree, (b) construct various types of constraints (e.g., to enforce the semantics of STRL operators), and (c) recursively construct an objective function doing a depth first traversal of the STRL tree.

We associate a binary decision variable I with each branch of the tree to allow the solver to cut or include entire subtrees. Setting an indicator variable to 1 for one of the MAX operator's children, for instance, effectively chooses a placement option encoded by that child. To capture the quantity of resource allocation from individual equivalence sets, we use partition decision variables. A given partition variable P_p^t encodes the amount of resource a solver will allocate from partition p at time t for duration dur of the nCk primitive processed. Constraints then build on indicator variables I and partition variables P_p^t to construct two main types of constraints. *Demand* constraints ensure that the sum of P_p^t over all partitions p that make up a given equivalence set equals to the requested k nodes. *Supply* constraints ensure that the sum of all partition variables P_p^t do not exceed capacity $cap(p,t)$ of partition p at time t. Lastly, the objective function for nCk is the value associated with the placement option it encodes, multiplied by the indicator variable I that governs the flow of value up from this primitive. It helps to visualize the objective function as the flow of value from its primitive leafs up the tree and aggregated at the root. On its way up, the flow is modified by various operators that channel the maximum flow, minimum flow, or aggregate flow from its children.

The intuition for KillnCk MILP generation is to contribute negative flow by returning $-c*I <= 0$, where c was the declared cost of preemption and I is the indicator variable. If chosen by the solver, the



(a) no preempt, hard constraint (b) no preempt, soft constraint (c) with preemption

Figure 6: Three jobs on a small heterogeneous cluster: 2 GPU machines (m1 and m2) on rack 1 and 2 non-GPU machines on rack 2. Job 1 (pink) arrives at t=0 and has no deadline (best effort). Job 2 (green) and Job 3 (blue) arrive at t=10 and have a deadline of t=35, indicated by the vertical red line; they also run faster if using preferred resources (both tasks on same rack for job 2, both tasks on GPU machines for job 3). JVuSched is able to meet all deadlines only with preemption. If jobs 2,3 wait for preferred resources or run on suboptimal allocation, they fail to meet their deadline. Deadline can be met only if job 1 is preempted.

given KillnCk's I variable will equal to 1, effectively choosing the preemption option encoded. KillnCk uses the same set of partition decision variables P_p^t, but uses them to encode the amount of capacity it will be contributing back to each of the partitions p at time t. Thus, KillnCk contributes a negative term to the aggregate supply constraint, indicating that, if chosen, the capacity held by this preemption option is contributed back to the corresponding set of occupied partitions. Lastly, the demand constraint enforces the all-or-nothing semantics of preemption. We make sure that the number of tasks preempted is exactly the number of tasks in this running job.

Finally, the Mixed Integer Linear Program portion contributed by KillnCk is recursively combined with MILP contributions from other parts of the aggregate STRL expression. It results in a single canonical MILP instance that includes contributions from all preemptable jobs as well as all pending jobs. The result of the MILP solver is a simultaneous determination of how much to preempt from each of the running jobs and which pending jobs to schedule instead—all done in a single scheduler cycle.

4.1.3 Preemption example

This section illustrates JVuSched's simultaneous cost/benefit consideration of preemption and placement with a simple example. Consider a small heterogeneous cluster and these 3 jobs: job 1, a best effort job, arrives and starts running at time 0. Job 2, an MPI job, can run in 20s if scheduled on the same rack or in 30s otherwise. Job 3, a GPU job, can run in 20s if scheduled on GPU nodes or in 30s otherwise. Jobs 2 and 3 are SLO jobs arriving at t=10 with a deadline at t=35. There are three options, all shown in Fig. 6. First, SLO jobs 2,3 may wait for their respective preferred resources to become available (Fig. 6(a)). Second, they may be scheduled on available resources as soon as possible (Fig. 6(b)), taking a performance hit and missing their deadline. Third, the scheduler may preempt the half-done best-effort job and simultaneously place SLO jobs 2,3 (Fig. 6(c)). This third option is the only way their deadlines can be met.

At the time when jobs 2,3 arrive at t=10, the resulting pSTRL expressions will be as follows.

- Running job1: $e_1 =$
  KillnCk({m2, m3}, k=2,startp=10,finishp=10,c=11)
- Pending MPI job2: $e_2 =$
  max(nCk({m1, m2}, k=2,start=10,dur=20,v=100),
  nCk({m3, m4}, k=2,start=10,dur=20,v=100),
  nCk({m1,m2,m3,m4}, k=2,start=20,dur=20,v=1))
- Pending GPU job3: $e_3 =$

```
max(nCk({m1, m2}, k=2,start=10,dur=20,v=100),
    nCk({m1,m2,m3,m4}, k=2,start=20,dur=20,v=1))
```

Thus, the resulting pSTRL expression to be maximized will be: sum(e_1, e_2, e_3). The MILP Compiler will then call the recursive MILP translation function gen(sum(e_1, e_2, e_3)) (Alg. 1, line 1). As the algorithm recurses down the expression tree to its leaves, it will process the killnCk primitive (line 1). Finally, the solver will determine that the highest possible value for this expression is $100 + 100 - 11$, where the negative term is contributed by the KillNck return call (line 1 in Alg. 1) and reflects the cost of preempting the running best-effort job. Upon completion of jobs 2,3 at t=30, the best-effort job is rescheduled again.

4.1.4 Greedy scheduling with preemption

JVuSched's preemption mechanism enables simultaneous consideration of pending jobs for allocation and running jobs for preemption. This is made possible by automatic translation of JVuSched's running and pending jobs into a single pSTRL expression in turn compiled into a single MILP instance and then solved. We refer to this simultaneity of preemption and allocation as *in situ* preemption and leverage it in our implementation of JVuSched's greedy mode. In greedy mode, JVuSched maintains two priority queues, one for SLO jobs and one for best effort. On each scheduling cycle, JVuSched selects the next SLO job in FCFS order and combines its STRL request with an aggregate pSTRL expression constructed for running jobs. The resulting expression's value is then maximized through the MILP solver, resulting in a set of running jobs to preempt and an allocation decision for the pending job considered.

4.2 Handling runtime under-estimates

Under-estimates can cause significant SLO violations in time-aware schedulers that depend on estimate accuracy. Once the scheduler detects that an under-estimate occurs, it has a choice to kill it [6] or to optimistically let it complete [28]. The latter was shown to significantly improve SLO attainment, but we have found that it must be kept in check because there can be large under-estimates. On every cycle, JVuSched performs the cost/benefit analysis to determine whether an under-estimated job should be allowed to continue. Given a near-perfect estimate, the best choice is to increase a job's expected runtime minimally and let it finish. For large under-estimates, however, it is ideal to discover and react quickly. We therefore implement an exponential back-off policy that increments the expected runtime $\hat{T}$ by 2^t cycles, starting with $t = 0$ incremented on each cycle. This achieves the desired effect of hysteresis in the system. JVuSched reacts to minor under-estimates with minor runtime estimate corrections. As it learns that the under-estimate is more significant, it updates the runtime by progressively longer increments.

The effect of such exponentially longer increments is three-fold. First, it increases the window of opportunity for the scheduler to preempt this job, effectively growing $finishp$ in Alg. 1. Second, given monotonically decreasing value functions for SLO jobs that reach zero past deadline, larger runtime estimate increments will eventually surpass the deadline, either triggering a kill event or increasing the probability of preemption by other jobs. Third, increased estimates cost more space-time in the scheduler's plan-ahead window. Running jobs always create an opportunity cost, which pending jobs may outweigh. Increased estimated space-time increases the probability that there exists a set of smaller jobs that may occupy the same amount of space-time and offer higher value for it. The combined effect is progressively increasing the likelihood that an under-estimated job will be either killed or preempted, achieving the desired effect of pruning vastly under-estimated jobs that may inappropriately consume valuable resources. In other words, JVuSched evaluates the net benefit of preemption of the remaining *estimated* space-time area of the late job. The key insight is that this remaining estimated space-time has a direct bearing on the scheduler's decision to preempt. Indeed, if a job is nearing completion, the benefit of preempting this job is far less significant, as the amount of space-time that frees is small relative to jobs with a much bigger estimate of remaining time to completion.

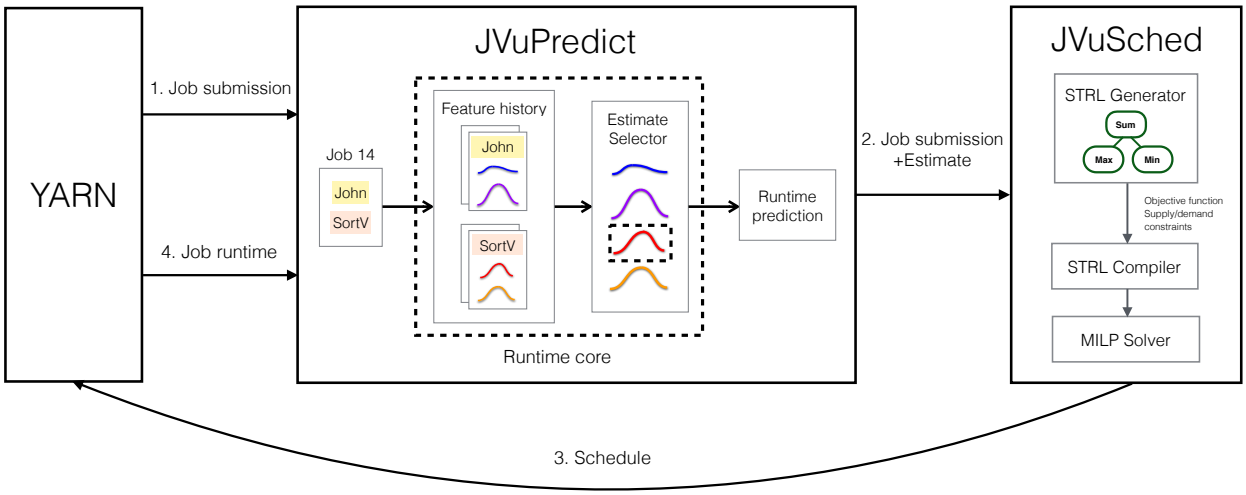


Figure 7: End-to-end system integration

4.3 Handling runtime over-estimates

In Sec. 4.2, we mention the possibility of SLO jobs' termination when their $\hat{T}$ is expected to exceed the deadline. To avoid unnecessary termination of jobs due to over-estimates, including over-estimates arising from adjusted under-estimates, JVuSched changes the job's value function to have a linearly decaying slope past the deadline, instead of the sharp drop to zero. The resulting effect is that over-estimated jobs may continue to run while their *expected* completion time exceeds the deadline. As the actual runtime nears the deadline, the job's preemption cost decreases, increasing the probability of preemption in favor of higher valued jobs. This naturally creates the desired effect of reducing resources spent on jobs that are increasingly likely to miss their deadline. With higher probability of the latter, the job's diminishing cost of preemption increases the probability of preemption and resource reclamation for higher valued pending jobs.

5 Experimental Setup

We conduct a series of end-to-end and partial system experiments to evaluate both JamaisVu's ability to predict runtimes and JVuSched's ability to absorb inherent mis-estimates. JVuSched is integrated into Hadoop YARN [29]—a popular open source cluster scheduling framework— as illustrated in Fig. 7.

Cluster Configuration We conduct experiments on two different cluster configurations: a 256-node real cluster (RC256), and a simulated 256-node cluster (SC256) for faster discrete-event simulation-based evaluation. For RC256, the experimental testbed [10] consists of 257 physical nodes (1 master + 256 slaves in 8 equal racks), each equipped with 16GB of RAM and a quad-core processor.

Workload Configuration All of our experiments use workloads derived from the Google cluster trace [21]. But, because we do not have access to a 12 thousand node cluster or the jobs themselves, we use the following process to generate a workload that will fit on our available clusters. To do so, we first filter out all jobs larger than 256 nodes. We use k-means clustering on the the remaining jobs' runtimes, associating each resulting cluster with a job class. We derive parameters for the statistical distributions of the job characteristics (e.g., runtime and number of tasks per job) in each job class. We also categorize the jobs by job attributes and, within each job class, we compute the distribution of features/attributes for jobs in that class. To maintain fidelity, we derive a probability mass function for the feature vector for each job class.

During the process of job generation from these job classes, our first step is to pick a job class, according to the relative weights (e.g., fraction of Google cluster trace jobs) of each job class. After picking the job class, we pick job characteristics and attributes from the job class according to the distribution observed in

SLO Miss %-age	Fraction of SLO jobs that miss their deadline
BE Goodput	total useful BE work (machine*hrs)
BE Latency	Mean response time for best effort jobs

Table 2: Evaluation Metrics

Workload	SLO Placement Constraint			Best Effort	Slowdown	Deadline Slack	Brief description
	Hard	Soft	None				
EXP1	25%	25%	0%	50%	1.5	[20, 40, 60, 80]	Workload with tightest deadline slack
EXP2	16.7%	33.3%	0%	50%	1.4	[25, 50, 75]	Workload with tighter deadline slack
EXP3	0%	50%	0%	50%	1.4	[50, 60, 70, 80]	Most lenient deadline
DEADLINE-n	0%	0%	50%	50%	N/A	[n]	Homogeneous Workload with fixed deadline slack *n* for all jobs

Table 3: Workload compositions used in results section.

the real trace.

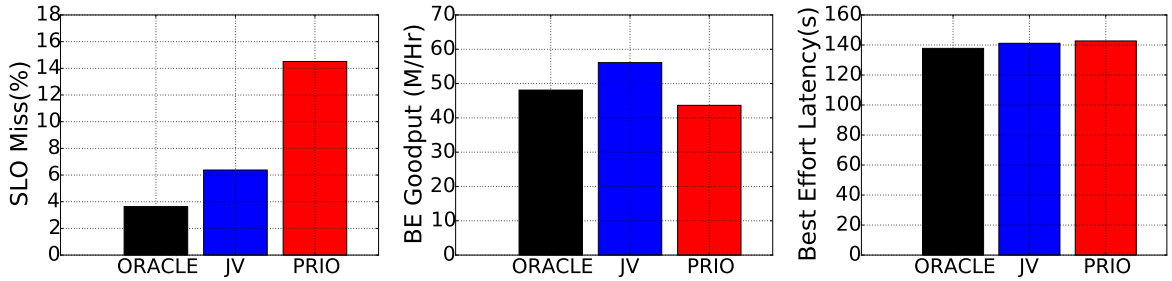
Another important system variable is the *deadline slack* perceived by the scheduler. Deadline slack is computed by $(deadline - submissiontime - runtime)/runtime * 100$ (i.e. a slack of 60% indicates that the scheduler has a window 60% longer than the runtime in which to schedule the job). As outlined in Table 3, we test our system against varying amount of slack.

Success Metrics. We use the following measures of success (Table 2). Our primary goal is to minimize the percentage of SLO jobs that fail to complete before their deadline, referred to as the "SLO Miss %-age". To account for best-effort latency sensitive jobs completed by the scheduler in addition to SLO jobs, we also calculate the BE goodput, which represents the BE work completed (calculated by summing up jobs' total machine-hours). BE goodput is a measure of cluster utilization, for BE jobs. Third, we measure mean best-effort latency. We aim to improve the performance for best-effort latency sensitive jobs as our tertiary objective.

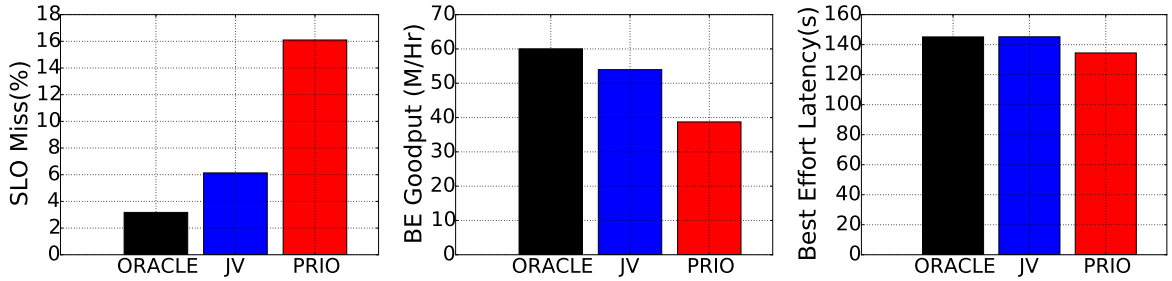
Systems under test (SUT). We configured JVuPredict to operate in multiple prediction modalities. JVuPredict can run in (1) full prediction mode or (2) "oracular" mode, when the runtime is perfectly predicted. The latter simply takes a job runtime estimate provided by the job submitter and is useful in trace-driven simulations, when the job's ground-truth runtime is known. We also compared JamaisVu to an implementation of Priority based scheduler – "prio", where the SLO job has strict priority over BE job.

6 Experimental Results

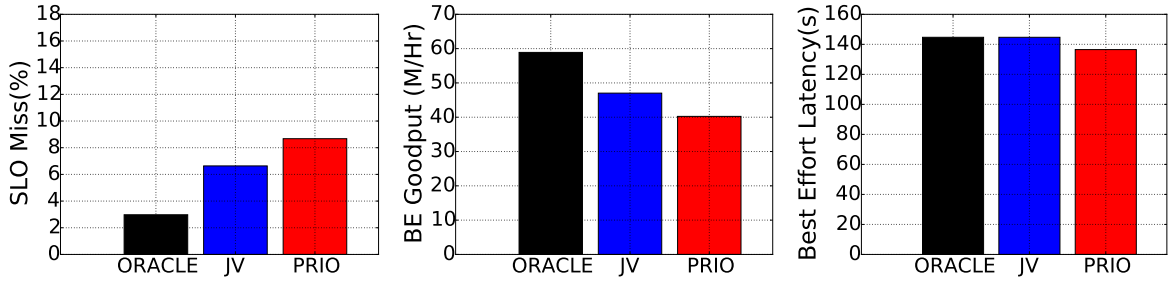
In this section, we evaluate our system's performance end-to-end, by deploying it on a real 256-node YARN cluster as well as testing it in simulation. Over numerous runs in simulation and the real 256-node cluster, we validated that the JVuSched core consistently returned an answer within the configured 5 second cycle period. Simulations routinely completed in $\frac{1}{5}^{th}$ the simulated time, thus averaging 1s scheduler cycle latency. This enables us to run multi-hour simulations in parallel, significantly increasing experimental throughput. There are X key experimental takeaways. First, we show that JamaisVu helps achieve significant improvements in SLO attainment, best-effort job goodput, and best-effort latency in a fully-integrated real cluster environment. These results are also echoed in simulation. Second, we show that JVuPredict achieves a narrow runtime mis-estimation profile for a set of jobs extracted from the Google cluster trace. Third, we demonstrate that JamaisVu performs well over a large range of relative SLO job deadlines. The deadline slack sensitivity analysis shows that the performance of our system converges to oracular, which is consistent with our intuition. Fourth, we break down individual contributions of three enabling JVuSched features : (a) "in situ" preemption support, (b) under-estimate handling, and (c) over-estimate handling. Together they constitute JVuSched's improvements over the best alternative baseline scheduler and help narrow the



(a) EXP1-SLO Miss rate (b) EXP1-BE Goodput (c) Exp1-BE latency



(d) EXP2-SLO Miss rate (e) EXP2-BE Goodput (f) EXP2-BE latency



(g) EXP3-SLO Miss rate (h) EXP3-BE Goodput (i) EXP3-BE latency

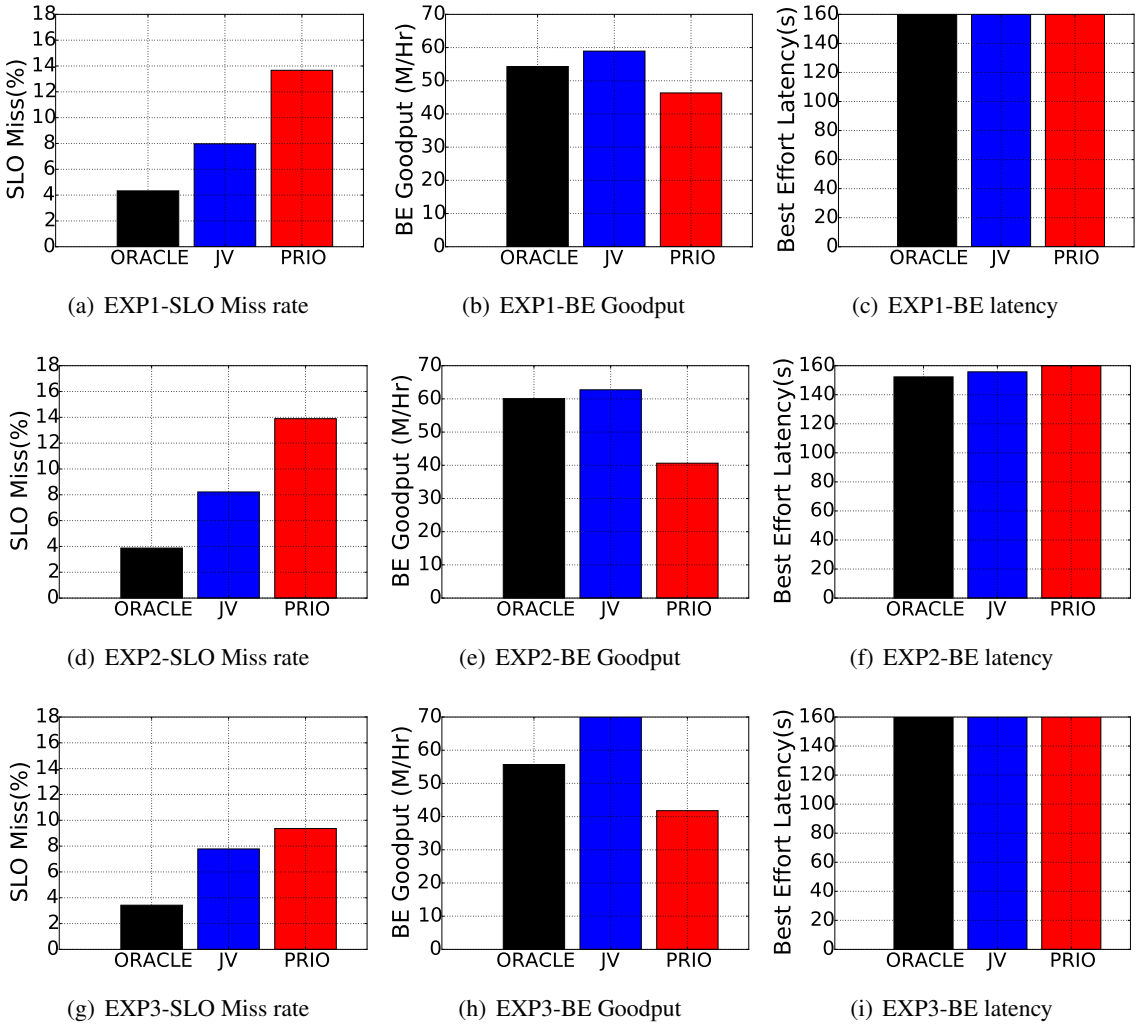
Figure 8: JamaisVu performance end-to-end on a real 256-node cluster (RC256). `JV` consistently outperforms strict priority scheduler on SLO miss rate and BE Goodput while nearly matching `oracle`. Similar to results with a simulated cluster in Fig. 9. Workload for EXPi described in Table 3.

performance gap with oracle.

6.1 End-to-end performance

First, we compare the performance of `JV` in RC256 with an implementation of the state-of- the-art strict priority scheduler `prio` designed to give SLO jobs strict priority over best-effort jobs. `Oracle` is the end-to-end system designed to use perfect runtime predictions. In Fig. 8(a) `JV` misses 8% SLO, much closer to `oracle` and making a significant departure from `prio`. The latter has 1.5x latency increase in best effort latency. By not taking advantage of resource predictions, `prio` is unable to achieve the utilization of `JV`. This causes the scheduler to miss more deadlines, despite scheduling SLO jobs first. `JV` closely matches `oracle` on all three metrics, yielding only slightly on the best-effort latency—our secondary objective.

We also validate our simulated environment SC256 that it produces similar result to real system using identical workload generated from same job characteristic and attribute distribution. As expected, we observe similar trends (Fig. 9) as we compare JamaisVu against `Oracle` and `prio`, in terms of SLO Miss rate, total BE goodput, and best-effort latency. Comparing with Fig. 8(a), we observe that the discrepancy is within 25%



(a) EXP1-SLO Miss rate (b) EXP1-BE Goodput (c) EXP1-BE latency

(d) EXP2-SLO Miss rate (e) EXP2-BE Goodput (f) EXP2-BE latency

(g) EXP3-SLO Miss rate (h) EXP3-BE Goodput (i) EXP3-BE latency

Figure 9: JamaisVu performance end-to-end on a simulated 256-node cluster (SC256). `JV` consistently outperforms strict priority scheduler on SLO miss rate and BE Goodput, while nearly matching `oracle`. Workload composition described in Table 3.

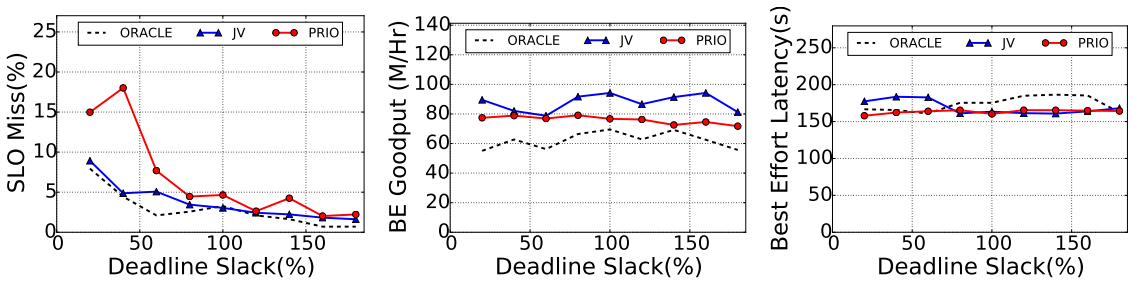
for SLO Miss, 14% for BE goodput, and 21% for BE Latency. However we note that simulated environment is less favorable for `JV`. That is `JV` performs more closely to `prio` away from `oracle` in the simulated environment compared to the real system.

6.1.1 Sensitivity Analysis

Fig. 10 compares JamaisVu's performance to `Oracle` and `prio` in SC256 as we vary the deadline slack. We report SLO miss rate, total BE goodput, and best-effort latency. We observe that JamaisVu's approaches to the performance of oracular predictor given sufficient deadline slack. Further, we note in Fig. 10(a) that increased deadline slack makes it easier to meet SLO deadlines, even with priority scheduler. The best-effort latency, our tertiary objective, does not exhibit any correlation with the chosen SUT and remains roughly the same for the tested range of deadline slack.

6.1.2 Applicability of JVuPredict in other systems

We have also integrated JVuPredict with the mainline YARN Rayon/CapacityScheduler stack. We observe that the relative performance differences between using oracular predictions and JVuPredict's estimates in YARN's default stack is comparable to that of JamaisVu. This confirms that JVuPredict can act



(a) SLO Miss-rate (b) Best Effort Goodput (c) Best Effort latency

Figure 10: JamaisVu outperforms a policy that explicitly prioritizes SLO jobs, approaching `oracle` in SLO misses, maximizes goodput, and minimizes latency for best effort jobs. Cluster:SC256, Workload: DEADLINE-n where $n \in [20, 40, 60, 80, 100, 120, 140, 160, 180]$

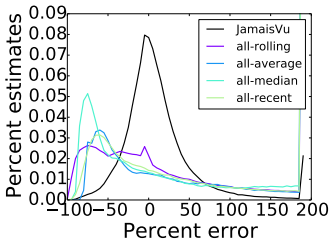


Figure 11: Prediction error mass function.

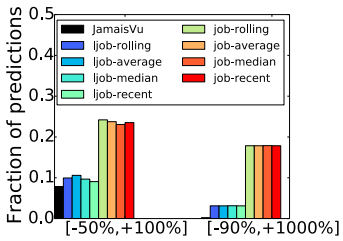


Figure 12: Predictions outside of given error ranges.

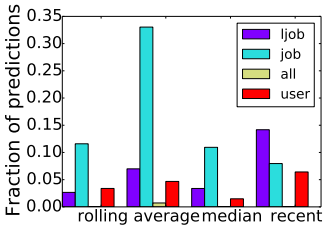


Figure 13: Feature:estimator selections for Google cluster trace.

as a general runtime estimator for different cluster schedulers.

6.2 Accuracy of JVuPredict

This section evaluates the overall accuracy of JVuPredict. To do so, we replay the entire Google cluster trace through the predictor, and compare JVuPredict's estimates with the job's traced runtime representing its actual runtime.

Fig. 11 shows the normalized prediction error mass function for JVuPredict, where each prediction's error is computed as $\frac{(Prediction-Actual)}{Actual}$. The large values at 100 (far right) represent all values $\geq$100. Also shown, for reference, is the prediction error that results from using each of JVuPredict's estimators with all previous runtimes, corresponding to a predictor that uses all history for every prediction rather than selecting subsets like JVuPredict does. While the "all" error profile is quite bad, JVuPredict's looks good visually: prediction errors are centered close to 0%, and most are relatively small. For example, 90% of job runtime estimates are within a factor of two (i.e., between -50% and +100%) of the corresponding actual runtimes. We subsequently define "bad estimates" to be estimates outside a factor of two of the actual runtime. Among these badly performing estimates, 11% are outliers and 1% have no history (thereby needing to rely on the all predictor). For 35% of estimates of the bad estimates, the best performing estimator had a normalized RMSE of less than 1.

Fig. 12 provides another view of the JVuPredict error profile, indicating the fraction of predictions that fall outside a given ranges (factor of two and a factor of ten of the runtime). In addition to the JVuPredict bars, bars corresponding to the most reliable individual feature:estimator options used within the predictor are shown. These bars confirm that JVuPredict's adaptive feature:estimator selection improves accuracy—for example, the percentage of prediction errors larger than a factor of two decreases from 12% (for "ljob-recent", the best-performing individual feature:estimator option) to 10%. Despite being generally reliable, predictors

using logical job and jobname both make a handful of extremely inaccurate estimates that are off by over a factor of ten. For these jobs, JVuPredict found a better estimate for all but 0.16% of jobs.

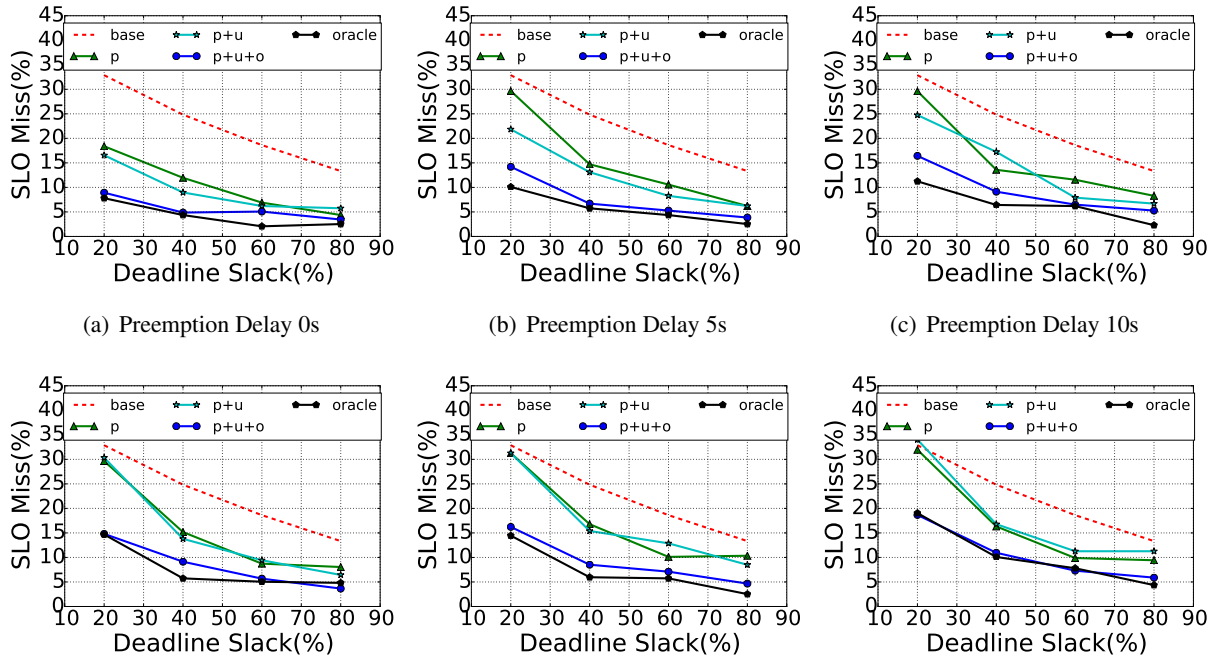
If only one feature:estimator pair is to be used, ljob-recent is the best choice. However, JVuPredict really does use its variety of options in practice, finding that different estimators are best for different jobs. The number of times each feature:estimator pair's prediction is selected by JVuPredict is shown in Fig 13. While job name is chosen the most, JVuPredict also sometimes chooses based on logical job name, user (e.g., because the user ID identifies a single-activity role), and even "all" (e.g., because there is no history for the job's other feature-values). We also observe diversity in the estimator policy selected for every feature-value, indicating the value of having them all.

For jobs clustered by user name, adding resources requested to the model as a "secondary" feature leads to a significant decrease in estimate error. However, this change has little effect on estimates made using other features. Consequently, using resources requested causes JamaisVu to choose a user-based estimate 3% more of the time. Given that user is not a frequently chosen estimator, this change leads to only a marginal improvement to the overall performance.

Introducing the option of a median filter (Sec 3.4) increases the number of estimates made within 10% by 2%. Overestimated errors of more than a factor of two also decrease by 2%.

6.3 Attribution of benefit

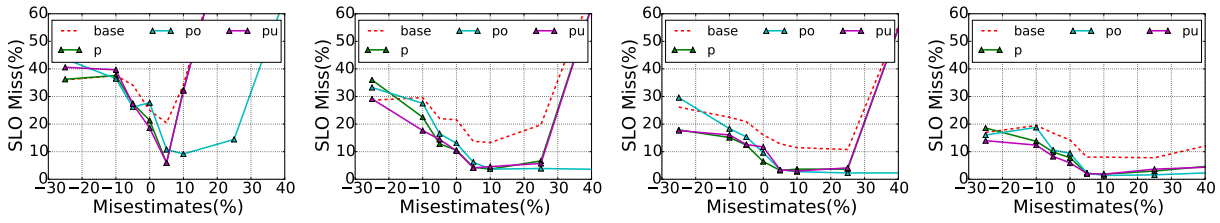
JVuSched introduces three main features to robustly address and absorb the effect of mis-predictions: (a) preemption, (b) under-estimate handling with exponential back-off, and (c) over-estimate handling, optimistically allowing jobs expected to miss their deadlines to start. This section breaks down the effect of these contributions on SLO miss rate and SLO work completed.



(a) Preemption Delay 0s (b) Preemption Delay 5s (c) Preemption Delay 10s

(d) Preemption Delay 15s (e) Preemption Delay 20s (f) Preemption Delay 30s

Figure 14: **Attribution of Benefit.** JVuSched clears the gap between state-of-the-art space-time aware scheduler [28] used in tandem with JVuPredict (`base`) and one using oracular knowledge of job runtimes (`oracle`). As we add preemption (p), under-estimate handling (u), and over-estimate handling (o) we significantly reduce SLO miss rate, clearing the gap between `base` and `oracle`. Cluster:SC256 Workload:DEADLINE-n where $n \in [20, 40, 60, 80]$



(a) Deadline Slack 20% (b) Deadline Slack 40% (c) Deadline Slack 60% (d) Deadline Slack 80%

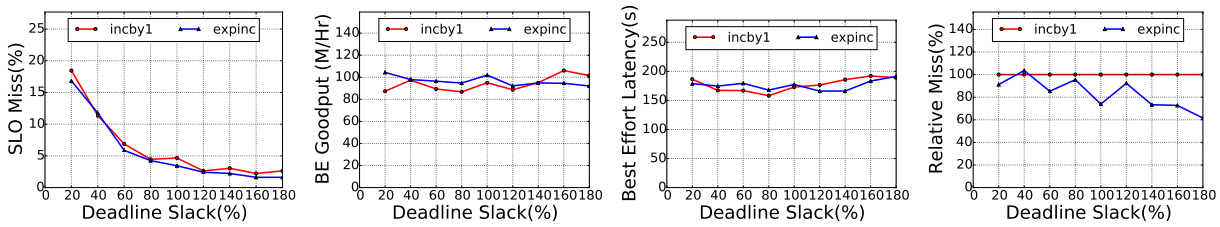
Figure 15: **Sensitivity to biased mis-estimation.** Benefits of under-estimation handling (pu), and over-estimation handling (po), as runtimes are respectively under- and over-estimated. `po` policy improves SLO miss rate for overestimates, `pu` policy improves SLO miss rate for underestimates. Both improve on `base`. Cluster:SC256 Workload:DEADLINE-n where $n \in [20, 40, 60, 80, 100, 120, 140, 160, 180]$

In Fig. 14, we vary the deadline factor on the x-axis, while measuring SLO miss rate as we vary the length of preemption delay of our end-to-end system in a simulation harness. We plot the performance of the barebones JVuSched using JVuPredict, but having all of its robustness features disabled (`base`), and the performance of JamaisVu as we turn on its robustness features one at a time. We also plot JVuSched operating with oracular predictor (`oracle`). We observe that enabling preemption (`p`) helps cut the SLO miss rate in half. Note that, as we increase the preemption delay, the benefit of preemption diminishes and all curves move closer to the `base`. The reason for this is fundamental: increased preemption delay reduces the agility of the scheduler in changing allocation decisions. When preemption delay is 0 (Fig. 14(a)), we see that all comparison curves are clustered in the lower half of the gap between the `base` and the `oracle`. As we increase the preemption delay, the `base` stays the same (it doesn't use any of these features), while the comparison curves gradually slide up the gap. This is especially visible at the lower end of the deadline slack range.

Adding under-estimate handling (`p+u`) improves on `p` slightly, particularly when the scheduler is more agile with its preemption (smaller preemption delay). A subsequent addition of the over-estimation handling feature (`p+u+o`), though, brings JVuSched's performance to match the oracle, which is the best possible outcome. The SLO miss rate (our primary objective) goes down to low single-digit percentage points across a range of deadline factors and with the real error profile induced by our online JamaisVu predictor trained on the Google cluster trace.

In Fig. 15, we control the amount of mis-estimation synthetically injected into the oracular predictor for a closer investigation of the effect of under- and over-estimation on our system. We illustrate the benefit of applying our targeted mis-estimation handling techniques to mitigate under- and over-estimates respectively and further show that both are needed for best performance across the spectrum of mis-estimates. Indeed, as the direction of mis-estimates is unknown and unpredictable in the real system, both mis-prediction techniques are simultaneously needed.

Lastly, in Fig. 16 we zero in on the benefits of exponential back-off. Intuitively, as the exact amount of mis-estimation is not known, we simultaneously wish to accomplish two things : (a) avoid over-reacting to very mild under-estimates by incrementing the runtime estimate minimally, and (b) quickly make the under-estimated job more amenable to preemption as soon as we discover that the under-estimate is more severe. These two objectives simultaneously tailor the system to both accurate predictors (effectively implementing hysteresis) and to worse predictors by adapting exponentially quickly.



(a) SLO Miss rate (b) BE Goodput (c) BE Latency (d) Relative SLO Miss

Figure 16: **Benefits of Exponential Back-off.** In nearly all cases, JamaisVu with `expinc` underestimate handling policy outperforms `incby1` in SLO miss rate and BE goodput, while maintaining comparable completion latency for best effort jobs. Cluster:SC256 Workload:DEADLINE-n where $n \in [20, 40, 60, 80, 100, 120, 140, 160, 180]$

7 Summary

JamaisVu is a new end-to-end cluster scheduling system that automatically generates and robustly exploits job runtime predictions on multi-purpose clusters. It uses a new black-box approach to generate predictions and several new techniques to mitigate the effects of the mis-predictions encountered in practice. Analysis of a month-long Google cluster trace shows JVuPredict's predictions are good, but as expected, not perfect. Experiments with trace-derived workloads both on a real 256-node cluster and in simulation demonstrate that the accuracy of JamaisVu's prediction engine, JVuPredict, is sufficient and the inherent mis-estimates are robustly handled given JVuSched's new techniques. Overall, JamaisVu's end-to-end performance approaches that of a hypothetical scheduler with oracular job runtime information and significantly outperforms runtime-unaware scheduling.

References

[1] PerfOrator, 2015. http://research.microsoft.com/en-us/projects/perforator.

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

[3] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of the VLDB Endowment*, PVLDB, 2012.

[4] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, 2011.

[5] Byung-Gon Chun. Deconstructing Production MapReduce Workloads. In *Seminar at: http://bit.ly/1fZOPgT*, 2012.

[6] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 2:1–2:14, New York, NY, USA, 2014. ACM.

[7] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 499–510, Berkeley, CA, USA, 2015. USENIX Association.

[8] Dror G Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling—a status report. In *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer, 2004.

[9] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 99–112, 2012.

[10] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 38(3), June 2013.

[11] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems (TOCS)*, 15(3):253–285, August 1997.

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[13] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[14] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: Towards a Scalable Workflow Management System for Hadoop. In *SWEET Workshop*, 2012.

[15] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[16] S. Krishnaswamy, S. W. Loke, and A. Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4), April 2004.

[17] David A Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer, 1995.

[18] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of mapreduce pipelines. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 681–684. IEEE, 2010.

[19] I. A. Moschakis and H. D. Karatza. Performance and cost evaluation of gang scheduling in a cloud computing system with job migrations and starvation handling. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 418–423, June 2011.

[20] John K Ousterhout. Scheduling techniques for concurrent systems. In *International Conference on Distributed Computing Systems (ICDCS)*, volume 82, pages 22–30, 1982.

[21] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of the 3nd ACM Symposium on Cloud Computing*, SOCC '12, 2012.

[22] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proc. of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 3:1–3:14. ACM, 2011.

[23] Ozan Sonmez, Nezih Yigitbasi, Alexandru Iosup, and Dick Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 111–120, New York, NY, USA, 2009. ACM.

[24] Roshan Sumbaly, Jay Kreps, and Sam Shah. The Big Data Ecosystem at LinkedIn. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2013.

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

[26] Alexey Tumanov, James Cipar, Michael A. Kozuch, and Gregory R. Ganger. alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proc. of the 3nd ACM Symposium on Cloud Computing*, SOCC '12, 2012.

[27] Alexey Tumanov, Timothy Zhu, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. TetriSched: Space-time scheduling for heterogeneous datacenters. Technical Report CMU-PDL-13-112, Carnegie Mellon University, Nov 2013.

[28] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.

[29] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, , Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of the 4th ACM Symposium on Cloud Computing*, SOCC '13, 2013.

[30] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[31] S. Verboven, P. Hellinckx, F. Arickx, and J. Broeckhove. Runtime prediction based grid scheduling of parameter sweep jobs. In *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pages 33–38, Dec 2008.

[32] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 48–56, Sept 2014.

[33] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.

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

[35] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007.

[36] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems (Eurosys)*, pages 265–278. ACM, 2010.

[37] Yanyong Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, March 2003.