Morpheus: Towards Automated SLOs for Enterprise Clusters

Sangeetha Abdu Jyothi [m,u] Carlo Curino [m] Ishai Menache [m]
Shravan Matthur Narayanamurthy [m] Alexey Tumanov [m,c] Jonathan Yaniv [t]
Ruslan Mavlyutov [m,f] Íñigo Goiri [m] Subru Krishnan [m] Janardhan Kulkarni [m]
Sriram Rao [m]

[m] Microsoft, [u] University of Illinois at Urbana–Champaign, [c] Carnegie Mellon University
[t] Technion-Israel Institute of Technology, [f] University of Fribourg

Abstract

Modern resource management frameworks for large-scale analytics leave unresolved the problematic tension between high cluster utilization and job's performance predictability—respectively coveted by operators and users. We address this in *Morpheus*, a new system that: 1) codifies implicit user expectations as explicit Service Level Objectives (SLOs), inferred from historical data, 2) enforces SLOs using novel scheduling techniques that isolate jobs from sharing-induced performance variability, and 3) mitigates inherent performance variance (e.g., due to failures) by means of dynamic reprovisioning of jobs. We validate these ideas against production traces from a 50k node cluster, and show that *Morpheus* can lower the number of deadline violations by 5× to 13×, while retaining cluster-utilization, and lowering cluster footprint by 14% to 28%. We demonstrate the scalability and practicality of our implementation by deploying *Morpheus* on a 2700-node cluster and running it against production-derived workloads.

1 Introduction

Commercial enterprises ranging from Fortune-500 companies to venture-capital funded startups are increasingly relying on multi-tenanted clusters for running their business-critical data analytics jobs. These jobs comprise of multiple tasks that are run on different cluster nodes, where the unit of per-task resource allocation is a container (i.e, a bundle of resources such as CPU, RAM and disk I/O) on an individual machine. From an analysis of large-scale production workloads, we observe significant variance in job runtimes, which sometimes results in missed deadlines and negative business impact. This is perceived by users as an *unpredictable* execution experience, and it accounts for 25% of (resource-provisioning related) user escalations in Microsoft big-data clusters.

Unpredictability comes from several sources, which for discussion purposes, we roughly group as follows:

- *Sharing-induced* – performance variability caused by inconsistent allocations of resources across job runs—a scheduling policy artifact.
- *Inherent* – performance variability due to changes in the job input (size, skew, availability), source code tweaks, failures, and hardware churn—this is endemic even in dedicated and lightly used clusters.

Unpredictability is most noticeable to users who submit *periodic* jobs (i.e., scheduled runs of the same job on newly arriving data). Their recurrent nature prompts users to form an expectation on jobs' runtime performance as well as react to any deviation from it, particularly, if the job is business-critical (i.e., a production job).

Unfortunately, widely deployed resource managers [9, 27, 51, 55] provide limited mechanisms (e.g., fairness weights, priorities, job killing) for users to cope with unpredictability of such jobs. Given these basic tools, users resort to a combination of ad-hoc tricks, often pivoting around conservative over-provisioning for important production jobs. These coarse compensating actions are manual and inherently error-prone. Worse, they may adversely impact cluster *utilization*—a key metric for cluster operators. Owing to the substantial costs involved in building/-operating large-scale clusters, operators seek good return on investment (ROI) by maximizing utilization.

Divergent predictability and utilization requirements are poorly handled by existing systems. This is taxing and leads to tension between users and operators.

An ideal resource management infrastructure would provide predictable execution as a core primitive, while achieving high cluster utilization. This is a worthwhile infrastructure to build, particularly, because periodic, production jobs make up the majority of cluster workloads, as reported by [43] and as we observe in §2.

In this paper, we move the state of the art towards this ideal, by proposing a system called *Morpheus*. Building *Morpheus* poses several interesting challenges such as, automatically: 1) capturing user predictability expectations, 2) controlling sharing-induced unpredictability, and 3) coping with inherent unpredictability. We elaborate on these challenges next.

Inferring SLOs and modeling job resource demands. Our first challenge is to formalize the *implicit* user predictability expectation in an *explicit* form that is actionable for the underlying resource management infrastructure. We refer to the resulting characterization as an (inferred) Service Level Objective (SLO). We focus on completion time SLOs or deadlines. The next step consists of quantifying the amount of resources that must be provisioned during the execution of the job to meet the SLO without wastefully over-provisioning resources. Naturally, the precise resource requirements of each job depend on numerous factors such as function being computed, the degree of parallelism, data size and skew.

The above is hard to accomplish for arbitrary jobs for two reasons: 1) target SLOs are generally unknown to operators, and often hard to define even for the users—see §2, and 2) automatic provisioning is a known hard problem even when fixing the application framework [52, 15, 26, 19]. However, the periodic nature of our workload makes this problem tractable by means of history-driven approaches. We tackle this problem using a combination of techniques: First, we statistically derive a target SLO for a periodic job by analyzing all inter-job data dependencies and ingress/egress operations (§ 4). Second, we leverage telemetry of historical runs to derive a job resource model—a time-varying skyline of resource demands. We employ a Linear Programming formulation, that explicitly controls the penalty of over/under provisioning—balancing predictability and utilization (§5). Programmatically deriving SLOs and job resource model enables a *tuning-free* user experience, where users can simply sign-off on the proposed contract. Users may alternatively override any parameter of the inferred SLO and the job resource model, which becomes binding if accepted by our system.

Eliminating sharing-induced unpredictability. Our second challenge is to enforce SLOs while retaining high-utilization in a shared environment. This consists of controlling performance variance with minimal resource over-provisioning. As noted above, sharing-induced unpredictability is a scheduling artifact. Accordingly, we structurally eliminate it by leveraging the notion of *recurring reservation*, a scheduling construct that isolates periodic production jobs from the noisiness of sharing. A key property of recurring reservations is that once a periodic job is admitted each of its instantiations will have a predictable resource allocation. High-utilization is achieved by means of a new online, planning algorithm (§ 6). The algorithm leverages jobs' flexibility (e.g., deadline slack) to pack reservations tightly.

Mitigating inherent unpredictability. Our last challenge is dealing with inherent performance variance (i.e., exogenous factors, such as task failures, code/data changes, etc.). We do this by dynamically re-provisioning the current instance of a reservation, in response to job resource consumption, in relationship to the SLO. This compensates for short-term drifts, while continuous retraining of our SLO and job resource model extractors captures long-term effects. This problem is in spirit similar to what was proposed in Jockey [19], as we discuss in §7.

We emphasize that all of the above techniques are framework-independent—this is key for our production clusters as they support multiple application frameworks.

Experimental validation. We validate our design by implementing *Morpheus* atop of Hadoop/YARN [51] (§8). We then perform several faithful simulations with traces of a production cluster with over 50k nodes, and show that the SLOs we derived are representative of the job's needs. The combination of tight job provisioning, reservation packing, and dynamic reprovisioning allows us to achieve: 5× to 13× reduction in potential SLO violations (with respect to user-defined static provisioning), and identical cluster utilization. All while, our packing algorithms leverage the flexibility in target SLOs to smooth the provisioning load over time, and achieve better ROI, by reducing cluster footprint by 14% to 28%. We conclude by deploying *Morpheus* on a 2700-node cluster, and performing stress-tests with a production-derived workload. This confirms both the scalability of our design, and the practicality of our implementation (§ 9). We intend to release components of *Morpheus* as open-source and the progress can be tracked at [2].

2 Motivation

In the early phases of our project, we set out to confirm/deny our informal intuitions of how big-data clusters are operated and used. We did so by analyzing four data sources: 1) execution logs of millions of jobs running on clusters with more than 50k nodes, 2) infrastructure deployment/upgrade logs, 3) interviews, discussion threads, and escalation tickets from users, operators and decision makers, and 4) targeted micro-benchmarks. We summarize below the main findings of our analysis.

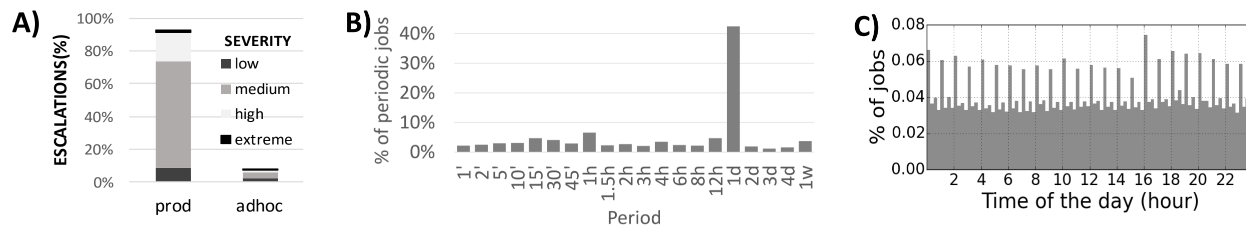

Figure 1: Analysis of user escalations and recurrent behaviors of production workloads.

2.1 Cluster workloads

Proper execution of production jobs is crucial. Production jobs represent over 75% of our workload and a similar percentage of the provisioned capacity—the rest being dedicated to ad-hoc jobs (10-20%) and ready to handle growth/failures (5-10%). All unassigned capacity is redistributed fairly to speed up jobs. As expected, users care mostly about proper execution of production jobs. Fig. 1a shows that over 90% of all escalations relate to production jobs, and this percentage grows to 100% for high/extreme severity escalations.

Predictability trumps fairness. Further analysis of the escalations of Fig. 1a and of discussion threads, indicates that users are 120× more likely to complain about the performance (un)predictability (25% of all job/resource-management escalations) than about fairness ($< 0.2\%$), despite the fact that our system does not enforce fairness strictly. This outcome may be expected, as customers cannot observe how "fair" allocations really are.

Production jobs are often periodic. Over 60% of the jobs in our larger/busier clusters are recurrent. Most of these recurring jobs are production jobs operating on continuously arriving data, hence are periodic in nature. Fig. 1b shows the distribution of the period for periodic jobs. Interestingly, most of the distribution mass is contributed by a small number of natural values (e.g., once-a-day, once-an-hour, etc.); this property will be useful to our allocation mechanisms (§6). Fig. 1c provides further evidence of recurrent behavior, by showing that job start times are more densely distributed around the "start-of-the-hour". This confirms that most jobs are submitted automatically on a fixed schedule.

The above evidence confirms that the most important portion of our workloads is strongly recurrent. This allows for planning the cluster agenda, without being overly conservative in the resource provisioning of jobs.

2.2 Predictability challenges

Manual tuning of job allocation is hard. Fig. 2a shows the distribution of the ratio between the total amount of resources provisioned by the job's owner and the job's actual resource usage (both comparing peak parallelism and area). The wide range of over/under-allocation indicates that it is very hard for users (or they lack incentives) to optimally provision resources for their jobs. We further validate this hunch through a user study in [15]. The graphs shows that 75% of jobs are over-provisioned (even at their peak), with 20% of them over 10× over-provisioned. This is likely due to users statically setting their provisioning for a periodic job. We confirm this, by observing that in one-month period over 80% of periodic jobs had no changes in their resource provisioning. Large under-provisioned jobs partially offset the impact of over-provisioning on cluster utilization.

Sources of performance variance. It is hard to precisely establish the sources of variance from the production logs we have. We observe a small but positive correlation (0.16) between the amount of sharing (above provisioned resources) and job runtime variance. This indicates that increased sharing affects runtime variance.

We investigate further the roles of sharing-induced and inherent performance variance by means of a simple micro-benchmark. Fig. 2b shows the normalized runtime of 5 TPC-H queries[1]. We consider two configurations one with constrained parallelism (500 containers), and one with unconstrained parallelism (>2000 containers); each container is a bundle of <1core,8GB RAM>. Each query was run 100 times in each configuration on an empty cluster at 10TB scale. The graph shows that even when removing common sources of inherent variability (data availability, failures, network congestion), runtimes remain unpredictable (e.g., due to stragglers, §7).

By analyzing these experiments and observing production environments, we conclude that: 1) history-based approaches can model well the "normal" behavior of a query (small box), 2) handling outliers (as in the long whiskers) without wasting resources requires a dynamic component that performs reprovisioning online, and 3) while each source of variance may be addressed with an ad-hoc solution, providing a general-purpose line of defense is paramount— see §7 for our solution.

[1]Box shows [25th,75th] percentiles, and whiskers shows [min,max].

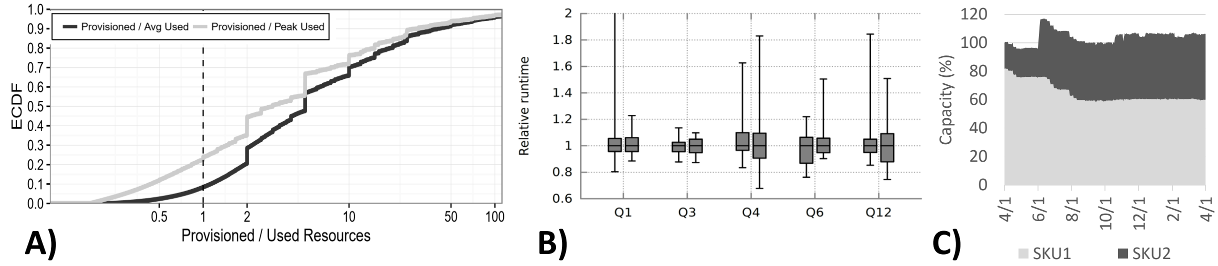

Figure 2: A) Empirical CDF of provisioning vs. used resources, B) box-whisker plot of normalized runtime of TPC-H queries running with 500 containers (left) and >2000 containers (right). C) cluster capacity of different machine types.

2.3 Changing conditions

Cluster conditions keep evolving—jobs may run on different server types. We provide in Fig. 2c a measure of hardware churn in our clusters. We refer to different machines configurations as Stock Keeping Units (SKUs). Over a period of a year, the ratio between number of machines of type SKU1 and type SKU2 changed from 80/20 to 55/45; the total number of nodes also kept changing over that period. This is notable, because even seemingly minor hardware differences can impact job runtime significantly— e.g., 40% difference in runtime on SKU1 vs SKU2 for a Spark production job.
User scripts keep evolving. We perform an analysis of the versioning of user scripts/UDFs. We remove all simple parameterizations that naturally change with every instantiation, and then construct a fuzzy match of the code structure. Within one-month of trace data, we detect that 15-20% of periodic jobs had at least one large code delta (more than 10% code difference), and over 50% had at least one small delta (any change that breaks MD5 of the parameter-stripped code). Hence, even an optimal static tuning is likely going to drift out of optimality over time.

Motivated by all of the above evidence, we focus on building a resource management substrate that provides predictable execution as a core primitive.

3 Overview of *Morpheus*

Morpheus is a system that continuously observes and learns as periodic jobs execute over time. The findings are used to economically reserve resources for the job ahead of job execution, and dynamically adapt to changing conditions at runtime. To give an informal sense of the key functionalities in *Morpheus*, we start our overview by following a typical life-cycle of a periodic job (JobX) as it is governed by *Morpheus* (§3.1). Next, we describe the core subsystems (§3.2). Fig. 3 provides a logical view of the architecture, and "zooms in" on a particular job.

3.1 "Life" of a periodic job

With reference to Fig. 3, a typical periodic jobs goes through the following stages.

1. The user periodically submits JobX with manually provisioned resources. In the meantime, the underlying infrastructure captures:
   (a) Data-dependencies and ingress/egress operations in the Provenance Graph (PG).
   (b) Resource utilization of each run (marked as the `R1-R4` skylines in Fig. 3) in a Telemetry-History (TH) database.
2. The SLO Inference performs an offline analysis of the successful runs of JobX:
   (a) From the PG it derives a deadline `d`—the *SLO*.
   (b) From the TH, it derives a model of the job resource demand over time, R^*. We refer to R^* as the *job resource model*
3. The user signs off (or optionally overrides) the automatically-generated SLO and job resource model.
4. *Morpheus* enforces SLOs via recurring reservations:
   (a) Adds a recurring reservation for JobX into the cluster agenda—this sets aside resources over time based on the job resource model R^*.
   (b) New instances of JobX run within the recurring reservation (dedicated resources).
5. The Dynamic Reprovisioning component monitors the job progress online, and increases/decreases the reservation, to mitigate inherent execution variability.
6. Morpheus constantly feeds back into Step 2 the PG and telemetry information of the new runs for continuous learning and refinement of the SLO and the job resource model.

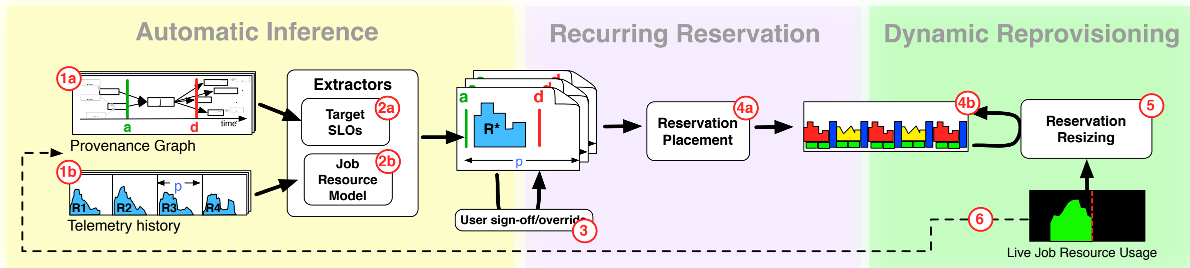

Figure 3: Conceptual view of *Morpheus*' architecture. Numbers/letters match the "Life" of a periodic job (§3.1).

3.2 Key components

We next give a brief overview of the different components in *Morpheus*, highlighting the different timescales in which they operate. Details follow in §4–§7. We start our description with the **automatic inference module**, which consists of two sub-components:

Extractor of target SLOs (§4). This sub-component operates on a Provenance Graph (PG). This is a graph capturing all inter-job data dependencies, as well as all ingress/egress operations. The SLO extractor leverages the precise timing information stored in the PG to statistically derive a deadline for the periodic job—as time at which downstream consumers read a job's output[2].

Job resource model (§5). This sub-component takes as input detailed telemetry information on job runs. The information includes time-series of resource utilization ("skyline")—the amount of resources (represented as containers) used by the job at certain time granularity, typically one minute. Based on time-series of multiple runs, the sub-component constructs a tight envelope R^* around the "typical" behavior of the job. These time-series are also used to derive the period, P.

The automatic inference module outputs the SLO and the job resource model in the form of a *recurring reservation request* for a newly observed periodic jobs, and continuously refines existing ones on slow time scale (e.g., daily). The inferred SLO and job resource model feed the resource reservation component automatically, yet *Morpheus* also allows for **user SLO and job resource model sign-off/override**. More specifically, the job owners are given three options: 1) sign-off on the proposed SLO and job resource model as-is, 2) override any of the parameters based on further knowledge (e.g., the job will run on 10× more data starting tomorrow), or simply 3) reject the use of SLOs, in which case the job runs with standard fair-queueing semantics [51]. By signing off a recurring reservation the user approves the SLO, the initial job demand skyline, as well as it accepts a bounded (and configurable[3]) amount of runtime reprovisioning—more below.

Reservation Placement (§6). The SLO and the job resource model are expressed in terms of a recurring reservation request to a Reservation Placement mechanism. *Morpheus* implementation (§8) builds upon YARN's reservation system [51, 13], which we extend to accommodate periodic reservations. The allocation problem itself poses a substantial algorithmic challenge, as the goal is to "pack" efficiently online arriving jobs with different periods and arbitrary skylines. To address this challenge, we design a novel online packing algorithm specialized for periodic jobs. The algorithm exploits the jobs' flexibility (e.g., deadline slack) to compactly pack them, which leaves enough capacity for ad-hoc jobs. Our algorithm is incremental as it places new reservation requests, without modifying the allocation plan for other jobs.

Dynamic Reprovisioning (§7). Naturally, any tight and static capacity reservation cannot perfectly accommodate all job instances. To cope with dynamic variability in job execution (or "inherent unpredictability"), this component continuously monitors the rate of progress of the job with respect to the amount of reservation consumed. If progress appears slower/faster than expected, the component automatically adjusts the reservation, by tweaking the resources provisioned for this reservation. Notably, such black-box approach is framework-independent, which is key given the large amount of frameworks that run in our clusters.

3.3 Current limitations

Before we fully describe *Morpheus*, we briefly highlight some limitations of our system.

Control over globally-shared resources. *Morpheus* relies on the underlying resource management infrastructure

[2]Note that a small number of periodic jobs exhibit latency-sensitive behaviors (output consumed immediately). Our system handles those as a special case of a deadline with no slack.

[3]This is currently a system-wide parameter, but it could be easily evolved to be a per-job parameter if demanded by customers.

(Apache Hadoop/YARN in our current implementation) to enforce its decisions. As such, *Morpheus* can only enforce container-level resources (such as CPU/Memory), but lacks control over globally-shared resources (e.g., bandwidth on switches, DNS server). Resulting runtime variability is coped with via dynamic reprovisioning (§7).

Support of non-periodic jobs. *Morpheus* supports both periodic and non-periodic reservations, but does not automate the SLO and job resource model extraction for never-seen-before jobs. Recent literature has shown that job resource modeling can be performed a-priori (from query and input data only) for a given application framework [15, 42, 41, 56]—we discuss integration of these approaches in *Morpheus* in §8. SLO extraction for never-seen-before jobs remains an open problem.

Automatic SLAs. *Morpheus* provides an important building blocks towards, but does not aim at delivering full-fledged automated Service Level Agreements (SLAs). A full SLA typically includes a specification of the economic (or business) aspects of the provider-user agreement [57]. For example, it may include the cost of unit of resources, threshold of expected SLO attainment, legal/financial consequences of missing the target SLO, limits of how much dynamic-reprovisioning is allowed and charging consequences, etc. These aspects require further investigation beyond the scope of this paper.

4 SLO inference

In this section, we show how to automatically derive SLOs for periodic jobs based on inter-job dependencies.

Based on interviews with cluster operators and users, we isolate one observable metric which users care about: job *completion by a deadline*. Specifically, analyzing the escalation tickets, some users seem to form expectations such as: *"95% of job X runs should complete by 5pm"*. Other users are not able to specify a concrete deadline, but do state that other teams rely on the output of their job, and may need it in a timely manner. Overall, the goal of *Morpheus* is to crystallize what users perceive as "good-enough" job performance through automatically-generated target SLO. Towards that end, *Morpheus* utilizes a Provenance Graph (PG) as the main inference tool. We next briefly describe the inference procedure.

Provenance Graph – reasoning about cluster data. The PG gathers logs (petabytes daily across our production environments) capturing key aspects of job execution, file system accesses, and system metrics. The PG is a semantically rich and compact (few TBs) graph representation of these raw logs. Specifically, *nodes* represent jobs and files in our clusters, and *edges* capture read/write operations among jobs, files, and all ingress/egress operations

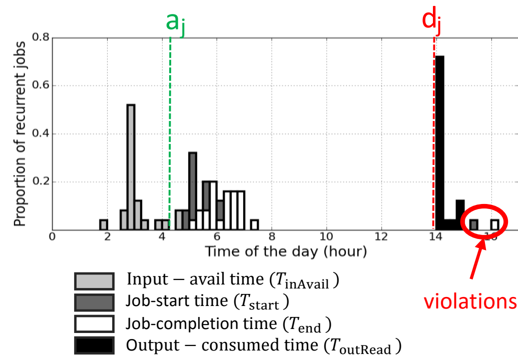

Figure 4: A periodic job from production traces.

(modeled as virtual source/sink nodes). This representation gives us a unique vantage point with nearly perfect close-world knowledge of the meaningful events in the cluster. The PG is constructed by scanning three sets of logs: application logs, filesystem logs, frontend logs. The application logs capture all job-related events such as: start and completion times, failures, and a job's inputs/outputs (this is part of the algebraic representation of the user query). The filesystem logs provide metadata information about files (size, nodes storing each block, etc.). The frontend logs capture upload/download operations from the cluster (i.e., ingress/egress). A daily batch job is used to parse the logs and by means of template-matching extract both the structure and node/edge properties which are then efficiently stored in the PG [37].

Isolating periodic jobs. We group individual job instances in a periodic job, if the templatized job names are an exact match, if source-code signatures are an approximate match, and if submissions have a near-constant inter-arrival time. The latter criterion is evaluated using the coefficient of variation (CV) measure of inter-arrival times. CV is computed as the ratio of median absolute deviation (MAD) (a robust estimate of dispersion) and central value (median), namely $CV = \frac{MAD}{median}$; we filter out jobs with large CV. We derive the period P_j of a job j based on the submission times—not subject to queuing delays.

Estimating SLOs from the PG. With reference to Fig. 4, our goal is to derive estimates for the earliest start time a_j and the deadline d_j for the job. To this end, we rely on four random variables, in chronological order: T_{inAvail}, time at which job inputs are available (i.e., the time of last write to any input); T_{start}, time when the job starts execution; T_{end}, time when the job completes execution; T_{outRead}, time at which any job output is first read. All these times are defined relative to the start of the current period $T_{\text{periodStart}}$[4]. We say that a job has an *actionable deadline* if its output

[4]The start time of the period of the ith job instance is given by $T_{\text{periodStart}} = \text{AbsoluteReferenceTime} + i \cdot \text{period}$, where AbsoluteReferenceTime is the time of first event recorded for the periodic job.

is consumed at an approximately fixed time, relative to the start of the period (e.g., everyday at 4pm), and if there is non-trivial amount of slack between the job end and the deadline. Formally this means imposing thresholds on $CV(T_{\text{outRead}})$ and median $\left(\frac{T_{\text{outRead}}-T_{\text{end}}}{T_{\text{end}}-T_{\text{start}}}\right)$. Finally, a_j and d_j are derived[5] as percentiles of the distributions of T_{inAvail} and T_{outRead} (e.g., 95th and 50th percentiles, respectively). The vast majority of periodic jobs in our workloads have actionable deadlines (§9), and will be offered an inferred SLO. The remainder will continue running with manually provisioned resources.

5 Job Resource Model

The second part of our inference module produces a resource allocation R^*_j that has high fidelity to the actual requirements of some periodic job j. In a nutshell, *Morpheus* collects resource usage patterns of periodic jobs over N_j instances that have run in the past, and solves an LP that "best fits" all patterns. Fig. 5 shows 4 runs (R1-R4) of a TPC-H query that were used, along with other runs, to generate R^*. The underlying optimization is governed by a parameter $\alpha \in [0,1]$ which determines the extent to which one wishes to reduce over-allocation of resources ($\alpha = 1$ hinting the maximal reduction of over-allocation). The usage patterns are captured as a set of skylines, one per run of a periodic job j. The resource allocation R^*_j is defined as the amount of resources to be provisioned (e.g., number of containers) at any point in time, for the successful execution of the different runs of j. For ease of presentation, we omit the index j, yet recall that all quantities below are for the same periodic job.

To derive the resource allocation, we first align the start times of all the job runs (instances), and quantize time, so that each quantized time-step corresponds to a fixed actual duration (e.g., one minute). Formally, a skyline for the i-th instance can be defined by the sequence $\{s_{i,k}\}$, the average number of containers used for each time-step k ($k \in 1,\ldots,K$). Using a collection of sequences as input, the optimization problem outputs the vector $\mathbf{s} = (s_1,\ldots s_K)$—the number of containers reserved at each time-step.

Our optimization objective is a cost function which is a linear combination of two terms: One term which penalizes for "over-allocation" $A_o(\mathbf{s})$, and another term which penalizes for "under-allocation" $A_u(\mathbf{s})$, both illustrated in Fig. 5; formally, we wish to minimize $\alpha A_o(\mathbf{s}) + (1-\alpha)A_u(\mathbf{s})$. Next we describe these terms.

[5]Note that, for a small fraction of the jobs, the periodicity of the job j can be smaller than the one of its consumers (e.g., daily jobs rolled up in a monthly report). In this case, we force a deadline based on the smallest periodicity to ensure the resource provisioning load is distributed over time (e.g., daily) instead of accumulated at the end (e.g., monthly). We confirmed with users that this aligns with their intents.

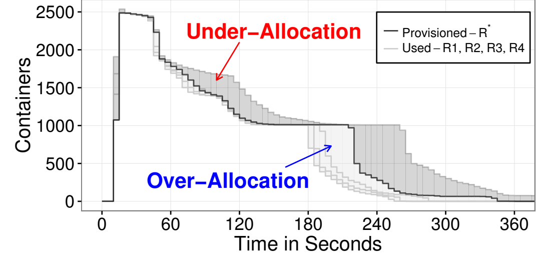

Figure 5: LP deriving a provisioned skyline R^*, from four runs (R1-R4) of TPC-H Query12 (10TB scale).

Over-allocation penalty. The over-allocation penalty is defined as the average over-allocation of containers. Formally, the expression $(s_k - s_{i,k})^+ = \max\{s_k - s_{i,k}, 0\}$ is the instantaneous over-allocation for instance i at time-step k. Accordingly, the over-allocation penalty is given by $A_o(\mathbf{s}) = \frac{1}{N}\sum_{i=1}^{N}\sum_k (s_k - s_{i,k})^+$.

Under-allocation penalty. We define a penalty which captures the *eventual* under-allocation of resources. Intuitively, we allow the job to "catch up" on under-allocations using resources available later in the run. Formally, we define the *debt* for instance i at time-step k as $D_{i,k}(s_1,\ldots,s_k) = (D_{i,k-1} + s_{i,k} - s_k)^+$, with $D_{i,0} = 0$. Observe that the allocation can decrease the debt over time, but cannot accumulate "credit" for later times (i.e., the debt cannot go below zero). The under-allocation penalty is the average debt at the last time step. Accordingly, $A_u(\mathbf{s}) = \frac{1}{N}\sum_{i=1}^{N} D_{i,K}(\mathbf{s})$.

The idea behind choosing these particular forms of penalties is to model, as closely as possible, the usage of allocated resources by a job that requests them. Particularly, the over-allocation penalty models the amount of unused resources because the job instance doesn't need them. Wasted resources allocated in a time-step cannot be recovered back at a later time-step. However, a shortage of resources at a time-step can be satisfied at a later point in time assuming the job is elastic. Final shortage of provisioned resources has to be counted only at the end; hence motivating the under-allocation penalty.

Avoiding lazy solutions. Just optimizing the above criteria can lead to solutions that lazily under-provision initially and compensate by aggressively allocating towards the end of a job's execution. So we add the following regularization constraint to the optimization problem $\frac{1}{N}\sum_{i=1}^{N}\frac{\sum_k (s_{i,k}-s_k)^+}{\sum_k s_{i,k}} \leq \varepsilon$. In words, we wish to sustain the average normalized instantaneous under-allocation below a threshold ε. While the objective and the constraints have non-linear terms, the optimization problem can be casted as an LP through standard lossless transformations.

The "right" value of ε may depend on the job characteristics (e.g., size, duration). In order to reduce the burden of calibrating the value of ε for every job, we roll ε into the optimization problem as follows. We add a linear term $\beta \cdot \varepsilon$ to the objective function. The value of β is set proportional to the other terms in the objective function, to make it relevant. Specifically, we solve the original optimization problem (without the $\beta \cdot \varepsilon$ term), and obtain a value V. We then set β to be a fraction of that value. Through experiments across many jobs, we found that setting β as $0.1V$ yields good results across the board.

Complexity. The LP has $(O(N \times K))$ number of variables and constraints. Our sampling granularity is typically one minute, and we keep roughly one-month worth of data. This generates less than 100K variables and constraints. A state-of-the-art solver (e.g., Gurobi, CPlex) can solve an LP of millions of variables and constraints in up to few minutes. Since we are way below the computational limit of top solvers, we obtain a solution within few seconds for all periodic jobs in our clusters.

Estimating parallelism. We assume that the skylines used to derive $\mathtt{R}^*$ are generated under capacity allocations sufficient to satisfy the maximum parallelism a job instance can harness. This assumption holds for production jobs because they are typically over-allocated to meet their deadlines. Under this assumption, we treat the estimate $\mathbf{s} = (s_1, \ldots s_K)$ of a job's resource requirement as also being its maximum parallelism for each timestep k. Further, we assume by default that the minimum parallelism of a job is one container (i.e., any requirement s_k can be stretched over time); this assumption can be overridden by either users or operators, assuming that they have additional knowledge about the inner working of the jobs. Inferring the min-parallelism automatically remains an open future direction.

6 Packing multiple periodic jobs

In this section, we provide an overview of *LowCost* – the algorithm we use to pack multiple periodic jobs.

6.1 Periodic reservations

Regardless of the packing algorithm we shall use, we face a practical challenge of how to reserve resources for multiple, possibly infinite, instances of a periodic job. It is inefficient to calculate and store a separate reservation for each instance of a periodic job. To address this challenge, we force the constraint that all instances associated with the same periodic job would have the same reservation across runs (namely, the same offset with respect to the period of the job). E.g., a daily job which requires 10 containers for one hour between 10am and 4pm maybe forced to execute between noon to 1pm every day. While this design choice might reduce the flexibility of a reservation-packing algorithm, it provides stronger predictability to users and reduces allocation complexity.

Having a fixed offset for each periodic jobs produces a repeating pattern in the overall allocation of all periodic jobs. We identify and store the smallest repeating unit which can accurately capture this recurring pattern in the set of all periodic jobs. In particular, we use the Least Common Multiple (LCM) of the time periods as the length of the internal storage unit. This ensures that all periodic jobs align with the boundaries of the storage unit; see Fig. 6 for an illustration. From an algorithmic perspective, one can determine how to pack multiple periodic jobs by only focusing on the LCM representation. This speeds up the packing algorithm, as it does not need to consider separately each instance of the periodic job.

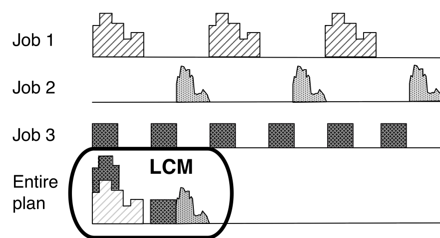

Figure 6: Illustration of LCM representation for multiple periodic reservations.

One may argue that the LCM can get very large, due to slightly "off-kilter" periods of a few jobs (e.g., 58 minute period). However, as shown in Fig. 1b, the distribution of periods in our clusters shows that most period values are divisors of one day. Accordingly, in practice, we set the LCM to be one day. The small fraction of jobs with periods that are not amenable (off-kilter or periods larger than one day) are accommodated using non-periodic reservations for each instance. We note that the LCM can be reconfigured in case of many such outliers.

6.2 Problem formulation

Setting. The input for a planning algorithm is a set of periodic jobs and a time range $[0, T]$, which represents the LCM period as described above. These jobs are typically revealed to the system one by one – i.e., in an online fashion. For simplicity, we describe the algorithmic problem under the assumption that each job has one instance within the LCM; we remove this assumption towards the end of the subsection. Each job j is characterized by a start time a_j, a deadline d_j, and a collection of stages $k \in [1, K_j]$. Each stage k captures a timestep of the reservation (see §5), hence is characterized by a total demand of s_k^j con-

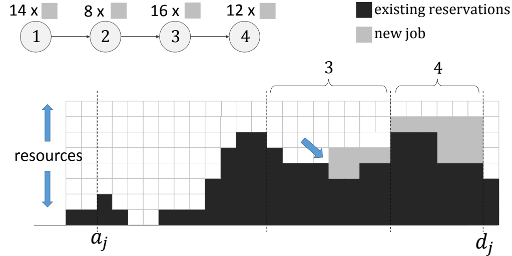

Figure 7: An example of *LowCost* execution. The new job has four stages with different number of containers. Stage 3 is currently being provisioned. Since the stage demands 16 containers and the total remaining demand is 38 containers, the time-interval for this stage is 16/38 of the time available, i.e., $\frac{16}{38} \cdot 19 = 8$. The arrow indicates where the next container of stage 3 would be allocated.

tainers; the stage may also have a minimum parallelism constraint (or gang size) of g_k containers.

Objective and Constraints. The goals of the packing algorithm are to (i) allocate containers to all periodic jobs, such that their requirements are met by the deadline, and (ii) minimize the waiting time for non-periodic, ad-hoc jobs. These goals can be better fulfilled if the cluster load is *balanced* over time. Intuitively, a balanced allocation increases the likelihood of accommodating future jobs (both periodic and non-periodic) that arrive into the system. As a concrete measure for a balanced allocation, the objective of *LowCost* is to minimize the maximal total allocation over time. We impose the following constraints on any solution. First, unless strictly necessary, we do not allow re-scheduling of jobs that are already in the system. This is important for business continuity. Second, because we typically use a sequence of stages to represent resource skylines, the entire allocation has to be *contiguous*, i.e., we do not allow "holes" in the allocation.

We note that the resulting online scheduling problem is hard already for single-stage jobs – Even the offline problem is NP-hard, as it generalizes the makespan minimization problem on multiple machines (e.g., [34]).

Requirements. We highlight the main requirements from a packing algorithm. The offline version of our planning problem can be casted as a Mixed Integer Linear Program (MILP). However, we prefer a quicker and "lighter" solution in terms of the running complexity. The main reason for not relying on rather costly solvers, is that *Morpheus* may often update the reservation plan. For example, upon arrival of a new periodic job, or as a consequence of changes in the resource estimations (hence reservation) of a job. On a related note, we need an *incremental* solution. That is, we wish to keep the reservations steady for jobs that are already in the system, and do not exhibit substantial changes in their resource demand.

6.3 Packing with *LowCost*

Cost function. *LowCost* uses a *cost-based* approach for allocation of containers that takes into account current cluster allocation and the resource demand of each job – each time slot t is associated with a cost $c(t)$. By default, the cost function $c : \mathbb{N} \to \mathbb{R}$ represents the current load of the cluster. Formally, $c(t) = \max\left\{\frac{\text{load}(MEM,t)}{\text{capacity}(MEM,t)}, \frac{\text{load}(cores,t)}{\text{capacity}(cores,t)}\right\}$, where $\text{load}(\cdot,t)$ represents the total allocation of the resource at time t, and $\text{capacity}(\cdot,t)$ represents its capacity.

The basic algorithm. In a nutshell, the idea behind *LowCost* is to allocate each incoming job in a way that is cost-efficient with respect to $\max_t c(t)$. To that end, *LowCost* follows a greedy procedure which places containers iteratively at cost-efficient positions.

In more detail, *LowCost* handles the stages one by one in reverse chronological order. For each stage k, *LowCost* first sets a time interval $I_{j,k} = [\tau^l_{j,k}, \tau^r_{j,k}]$ during which the stage can be allocated. $\tau^r_{j,k}$ is set right before the allocation of stage $k+1$. The length of $I_{j,k}$ is set proportional to the ratio between the demand of the stage and the total demand of the remaining stages, i.e., $\frac{s^j_k}{\sum_{k'=1}^{k} s^j_{k'}}$; see Fig. 7 for an example. To accommodate the contiguous allocation constraint, the eligible time steps for allocating the next gang of a given stage are $[\tau^{\text{cur}}_{j,k} - 1, \tau^r_{j,k}]$, where $\tau^{\text{cur}}_{j,k}$ is the leftmost timestep which includes some non-zero value for the current allocation to the stage. *LowCost* repeats the above procedure for different end points, and chooses the allocation with the minimum cost.

Multiple instances. Finally, a periodic job may have *multiple instances* within the LCM (e.g., an hourly job j, where the LCM is one day). As mentioned earlier, we place all the instances of the job with the same offset with respect to the period of the job (e.g., all instances of j should start at the same time-of-day). We incorporate this constraint in *LowCost* as follows. Observe that we essentially need to decide on the placement of a single instance. To do so, for each timestep within the job's period, we set the cost as the maximal cost across all timesteps with the same offset with respect to the period. For example, the cost seen by j at the 5-th minute would be the maximum over the costs at 12:05, 1:05, etc. *LowCost* then places a single instance based on these costs, and repeats the assignment for all instances within the LCM.

We wish to analyze in isolation the consequences of this choice. Accordingly, for the analysis sake, we assume that all periodic jobs have the same skyline requirement

for their instances, but still jobs can differ in their period. Under these assumptions, we measure the performance of *LowCost* using the standard measure of competitive ratio. The competitive ratio of an online algorithm is the ratio of cost (for our problem, the maximal height of the allocation) incurred by the online algorithm compared to the offline optimal solution. Let $P_{\min}, P_{\max}$ denote the minimum and maximum period of jobs within the LCM. We have the following guarantee:

Theorem 6.1 *Under the above assumptions,* LowCost *is* $O\left(\log\left(\frac{P_{\max}}{P_{\min}}\right)\right)$*-competitive for the objective of minimizing the maximum height of the allocation.*

The proof follows by showing that *LowCost* leads to an efficient (constant-competitive) schedule when jobs have the same period. The log-factor arises due to the range of possible job periods. Intuitively, this result implies that there is bounded performance loss due to the combination of our design and algorithmic choices.

Non-periodic jobs. So far we described how we reserve resources for periodic jobs. We now briefly address how *Morpheus* handles non-periodic jobs. The design of *Morpheus* assumes that periodic jobs have strict priority over non-periodic (mostly ad-hoc) jobs. This is commensurate with our analysis, which indicates that the bulk of periodic jobs are (business-critical) production jobs (see §2). Accordingly, when *Morpheus* needs to allocate resources to a new periodic job, it ignores most of the scheduled non-periodic jobs (excluding periodic jobs that are handled as non-periodic ones), and then attempts to reallocate resources for non-periodic jobs in case they need more resources. Specifically, *Morpheus* places the non-periodic using the same logic of the basic *LowCost* algorithm, described above. The only difference is that the plan for the lower-priority non-periodic jobs uses the residual capacity, after subtracting the chunk used for periodic jobs.

7 Dynamic Reprovisioning

While reservations can eliminate sharing-induced unpredictability, they provide little protection against inherent unpredictability arising from hard-to-control exogenous causes, such as *infrastructure* issues (e.g., hardware replacements (see §2), lack of isolation among tasks of multiple jobs, and framework code updates) and *job-centric* issues (changes in the size, skew, availability of input data, changes in code/functionalities, etc.).

Although eliminating all the causes of unpredictability is very hard, we can mitigate their impact on SLO attainment during runtime, by dynamically modifying the current instance of a periodic reservation. To that end, we design a *dynamic* reprovisioning mechanism which is triggered when a job execution appears to be headed for an SLO violation.

Dynamic Reprovisioning Algorithm (DRA). The Dynamic Reprovisioning Algorithm (DRA) we currently employ in *Morpheus* continuously monitors the resource consumption of the job, compares it with the resources allocated in the reservation and intuitively "stretches" the skyline of resources to accommodate a slower-than-expected job execution. Reprovisioning is triggered when a job resource demand (used containers plus pending ask) exceeds the resources allocated in the skyline. Extra resources are granted for up to T seconds (default 1min), after which DRA is reevaluated again. The amount of extra resources is based on the job's instantaneous demand, but capped at $\rho * \max(R_{\text{recent}})$ where R_{recent} is the amount of resources allocated in the skyline in the last few minutes (default 2min), and ρ is a fudge factor (default value 2) that allows an elastic job to use extra parallelism to make up for lost time; note that DRA verifies that the job does not get more resources than it requests. Given this proposed reprovisioning, DRA updates the current instance of the periodic reservation (by increasing it locally). This is done by invoking *LowCost*, which ensures the update is accepted only if enough resources exist in the plan.

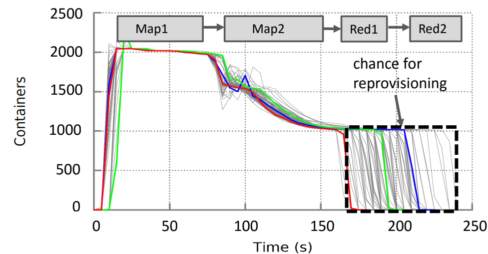

Figure 8: Resource consumption over time for 100 runs of TPC-H Query1 on 2200 parallel containers (job running alone in the cluster).

The proposed heuristics cope well with the inherent unpredictability we observed in Section 2.2. We show this by plotting in Fig. 8 the resource consumption over time for the TPC-H Query1 (100 runs, and highlighting 3 random ones in red/green/blue). DRA kicks in for jobs that have straggling Map2 tasks, which translates in a delayed start of Red1 stage. By extending the 1000 containers allocation at the end of Red1 by an extra minute we allow most jobs to complete effectively. Similar analysis has been performed for other TPC-H queries with equally good results, and in §9 we validate DRA performance on large production traces.

DRA is simple to implement and rather robust, however deeper understanding of the application-framework could lead to more precise reprovisioning decisions. In

Morpheus' pluggable architecture, this could be achieved by borrowing techniques from [15, 41, 19].

Adjusting *LowCost* to facilitate reprovisioning. The position of the original reservation allocation with respect to the $[T_{\text{inAvail}}, T_{\text{outRead}}]$ window is critical for the effectiveness of dynamic reprovisioning. In order to improve the success probability of reprovisioning, it is necessary to allocate resources far away from the deadline (allowing sufficient slack in time for the reprovisioning algorithm to compensate a slower than expected run). However, in case of high variance in input data availability, it is beneficial to place the allocation close to the deadline (to ensure that data is available before allocation and thus reduce the probability of reprovisioning). To account for this trade-off, we adjust *LowCost*'s cost function for "problematic" jobs (e.g., jobs with high CV for $T_{\text{start}} - T_{\text{inAvail}}$, $T_{\text{outRead}} - T_{\text{end}}$) by adding an *alignment penalty* . Specifically, the penalty is linearly proportional to the absolute time-distance between the mid-point of the allocation, and the mid-point between the start time and the deadline (i.e., $\frac{a_j+d_j}{2}$). This penalty incentivizes allocations that are not too close to neither the start or the deadline of the job. This trades the two dangers of allocating resources before the input is available, and not having enough slack after the allocation before the deadline.

8 Implementation

We implement the design of §3 as extensions to Apache Hadoop / YARN [51]. Referring back to the architecture of Fig. 3, we implement the three components of *Morpheus* as follows. First, the automatic inference engine operates as a standalone service. It continuously consumes provenance and telemetry data and submits reservation requests to the Resource Manager (RM)—YARN's centralized scheduler component [51]—via its REST endpoint. Second, the reservation placement component implements *LowCost* as in-process functionality of the RM. Third, the dynamic reprovisioning mechanism is implemented as a monitoring thread in the RM, which observes job resource requests and triggers resizing of reservations. Each of the above components is highly pluggable and can easily be specialized to leverage framework-specific knowledge, such as [19, 41, 15].

In the rest of this section, we discuss some of the engineering challenges in building a production-ready system.

Scalability. *Morpheus*' periodic reservations are instantiated as per-job queues in the RM. Each queue's capacity continuously grows and shrinks according to the provisioning skyline. YARN's RM scheduler [51], is designed to support a small number of infrequently reconfigured queues (e.g., one per division of a company). Hence, the implementation leveraged strict consistency via locking for queue updates. This limited *Morpheus*' scalability to levels far below our production needs. We address this by substantially reworking the RM scheduler locking mechanisms through a combination of finer-granularity locking and lock-free data structures. The key intuition is that the RM operates as an asynchronous event-driven system based on heartbeats and, therefore, is amenable to operating with relaxed consistency. We carefully study the effects of our changes and confirm that they induce very small and transient inconsistencies, that are naturally resolved without visible impact within milliseconds. This results in a sustained scalability orders of magnitude higher than the baseline. We showcase this experimentally running on a 2700-node cluster in §9.3.

Cold-Start. An obvious concern for a system that relies on history to make inference is how to handle cold-start scenarios, such as non-recurring jobs or initial runs of a new recurring job. We have three lines of defense to cope with this problem: *(a)Backward compatibility:* Our approach by design is able to support running jobs with existing fair-queueing infrastructure mode. *(b) Manual SLOs and job resource models:* The APIs supporting the sign-off (step 3) in our job lifecycle can be used to supply a manually defined SLO and job resource model (both for periodic and non-periodic jobs [13]). *(c) Application-specific tools:* Given a fixed application framework (e.g., Hive/Giraph/Scope/Spark) it is possible to build tools that leverage sample runs and careful modeling to predict the behavior of the full-scale execution of the jobs. We experimentally integrated with Predict [41] to support Giraph computations, as well as recently enabled similar functionalities for Hive/Tez/MapReduce with the Perforator [15] effort. In [15], we take Hive queries and perform cardinality estimation via lightweight profiling of UDFs. We then use this accurate cardinality estimates together with explicit models of Tez/MapReduce pipelining and parallelism and hardware performance profiles to estimate a job demand model. Perforator is integrated with our infrastructure, but complete integration between Morpheus and Perforator technologies is part of our future work.

9 Experimental Evaluation

In this section, we demonstrate effectiveness and scalability of *Morpheus* through simulations and cluster runs.

Experimental Settings Our experiments are based on two production traces and a synthetic benchmarking suite: *Enterprise-trace*, a one-month trace of jobs running on a large 50k-node production *COSMOS* cluster [9]; *Hadoop-trace*, a three-month trace derived from a 4k nodes production Hadoop cluster; *TPC-H*, the standard TPC-H

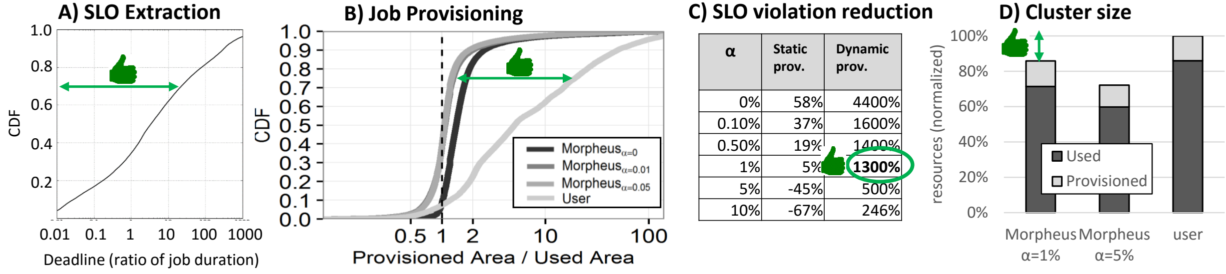

Figure 9: Comparison of *Morpheus* with current user manual provisioning.

benchmark running on Hive/Tez at 10TB scale. The enterprise-trace has been discussed in §2, and TPC-H is well documented [12]. The table below presents a breakdown of jobs types and size for our hadoop-trace. Jobs are clustered into multiple classes based on duration and size.

Hadoop-trace

framework	class	freq. %	avg duration (sec)	avg parall.
MR/TEZ	S	7%	73	1.5
	M	15%	156	19
	L	0.6%	2778	469
SPARK	S	39.8%	173	2.6
	M	14.52%	605	18
	L	7.8%	1400	88
	XL	4%	6300	510
	XXL	8.6%	24570	1000
MPI	-	1.56%	7800	400

For each category we extract salient statistical distributions: job arrival times, workload frequency, job parallelism, and job duration. These distributions are used to power a Gridmix-based [48] load generator.

9.1 Performance on the enterprise-trace

First, we challenge *Morpheus* in a simulation based on our largest dataset, the **enterprise-trace**. For this data-set we have full provenance graph (PG) and telemetry information, and we can thus test all components of *Morpheus*.

Sensible SLOs for most jobs. A pressing question we want to answer is whether the SLOs we derive are representative of user expectations. Short of a full-scale user study, we study a reliable proxy metrics: job success/-failure. Given job pairs $A \rightarrow B$, such that B is the first consumer of A's output, we measure from the trace:

$$P(B_{\text{fail}} \mid A_{\text{missSLO}}) \approx P(B_{\text{fail}} \mid A_{\text{fail}}) > 4 \times P(B_{\text{fail}} \mid A_{\text{meetSLO}})$$

This shows that the negative impact of missing a deadline is comparable with the impact of complete failure of the job. This is empirically 4× worse for the dependent job B than if A had met the SLO.

Second, we observe that *Morpheus* SLO target extractor successfully derives SLOs for over 70% of the millions of instances of periodic jobs in the enterprise-trace. For the remainder we have too little data in our trace to derive SLOs with good confidence (e.g., we only have four samples in our trace for jobs with weekly periodicity).

SLOs, job modeling, packing, and reprovisioning. In Fig. 9 we use our enterprise-trace (70% training and 30% testing) to show *Morpheus*' ability to: (A) extract SLOs, (B) derive job resource models, (C) achieve high SLO attainment gains over the baseline, and (D) pack reservation efficiently (measured as potential cluster reduction). In Fig. 9a, we present a CDF of the ratio between the slack (time between job-completion and deadline) and the job duration ($\frac{T_{\text{outRead}} - T_{\text{end}}}{T_{\text{end}} - T_{\text{start}}}$). The majority of jobs have substantial amount of slack (almost 70% of jobs have enough slack to serially execute two or more times before the deadline)—this flexibility is leveraged during packing. Fig. 9b compares the job provisioning achieved by *Morpheus* under different assignments of the parameter α with the user-supplied one (matching our motivation Fig. 2a). *Morpheus* drastically outperforms the user, by being consistently closer to the ideal provisioning (1:1 ratio, shown as vertical dotted line). Different assignment of α affect how tightly the skyline is fitted, but also how likely we are to miss an SLO (Fig. 9c). We find that a value of 1% leads to the best balance, yielding 13× reduction of the worst-case SLO misses—these are defined as the amount of SLO violations a periodic job would incur if no opportunistic (fair-share) capacity is available. Finally Fig. 9d shows that our packing algorithms manage to handle the complex skylines produced by the job modeling component, and leverage the slack in SLOs to densely pack the cluster agenda. This matches our important constraint of not increasing the cluster cost (but actually lowering it). The ratio between used and provisioned indicates that we achieve high-utilization, even though we rely solely on guaranteed provisioning, while the user compensate with under-allocation via opportunistic fair-sharing. Note that our unused capacity is anyway redistributed fairly, but we do not rely on it to achieve high utilization.

9.2 Breakdown of contributions

Fig. 10a shows a breakdown of contributions of our static techniques. We fix a target SLO attainment level 5×

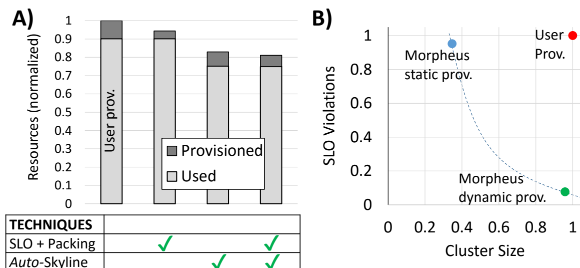

Figure 10: Gain break-down: each technique employed by *Morpheus* delivers sizable improvements.

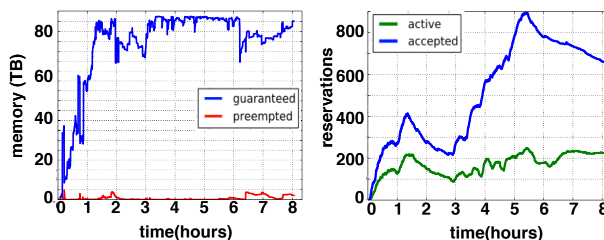

Figure 11: Scalability metrics for large scale *real cluster* run (on 2700 nodes)

above the baseline, and show the smallest size cluster required to achieve that under different combinations of our techniques. In particular, we show that: 1) SLO-extraction + packing lower the baseline cluster size by 6%, 2) Job resource modeling, i.e., using our skyline instead of user supplied provisioning, can alone lower cluster size by 16%, and that 3) when combined they achieve 19% total reduction. Fig. 10b highlights the trade-off between utilization and predictability, by showing how turning on/off our dynamic reprovisioning we can either: 1) match the user utilization level, and deliver 13× lower violations, or 2) match the current SLO attainment and reduce cluster size by over 60% (since we allow more aggressive tuning of the LP, and repair underallocations dynamically). Hence, each of the techniques we developed is required and supplies a substantial portion of our overall win.

9.3 Physical deployments and scale tests

In this section, we challenge *Morpheus* with a combination of the Hadoop-trace and the TPC-H workloads. We test our system under 3 environments: (a) 275-node cluster with 2200 containers (1core, 8GB RAM per container), (b) 2700-node cluster (100k containers) and (c) 4000-node production cluster.

Adversarial workload. We validate *Morpheus*' ability to protect jobs from an adversarial workload. In Fig. 12, we show an hourly periodic workload comprised of several TPC-H queries running on a 275-node cluster (equivalent to 2200 containers). The workload imposes heavy load on the cluster, with container utilization hovering near 100% of the available capacity during most of the experiment. The jobs are submitted periodically within a reservation we derived from historical runs. We then surround the periodic jobs with thousands of ad-hoc jobs from the Hadoop-trace which can take upto 75% of the cluster capacity. Thus, periodic reservations are run against a cluster stressed with production workload. *Morpheus* successfully eliminates all sharing-induced variability. To further challenge our system, we manually delay the start of one of the queries. The system immediately reacts by dynamically reprovisioning the (delayed) job with extra capacity, compensating for our actions and meeting the target SLO.

Scale test (2700 nodes). We validate *Morpheus*' scalability to target production clusters, by running it live on a *2700-node cluster*, scheduling almost 100k concurrent containers through the ResourceManager. This is a high-load workload designed to stress the scheduling infrastructure. We run a sustained 8hr experiment, with hundreds of reservation submissions per hour. We measure the system performance both as perceived by the user (not shown), and as observed by instrumented system components (Fig. 11). The key takeaway of this experiment is three-fold: 1) we demonstrate that *Morpheus* is able to sustain high load on a large cluster, 2) we confirm that in a real deployment *Morpheus* can achieve high plan utilization, 3) we confirm that user-facing latencies are in-line with production cluster user expectations. We see up to 900 concurrent reservations in the plan, with up to 270 of them active throughout the 8hr run. At peak, aggregate guaranteed capacity exceeds the 92TB of container memory, reaching maximum cluster capacity. The system remains responsive throughout the experiment with reservation submission latencies within 10sec.

Production deployment. We validate our system by deploying it in a 4000-node production environment. In this context, we are only allowed to run a small number of periodic jobs via reservations, while the bulk of the load is imposed by ad-hoc and manually provisioned jobs. Focusing on a periodic run of TPC-H Query3, the runtime variability was well controlled, despite utilization swings of whole cluster in excess of 69k cores during the job execution. During the same period, jobs running without the protection of reservations observed much larger variance.

10 Related Work

SLO extraction. To the best of our knowledge, we are the first to propose fully automated extraction of SLOs from historical data. Close related work focused on semi-automated, iterative generation of SLOs for databases [40] and web services [46].

Runtime/provisionining estimation. Substantial re-

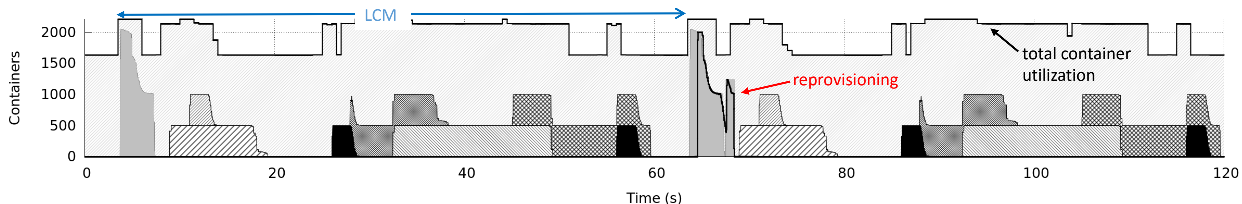

Figure 12: Run on 275-node cluster: shows an example of successful dynamic-reprovisioning.

lated work has been devoted both in database and systems literature to estimate query runtimes, and resource needs. Runtime prediction has been studied in databases [33, 10, 21], Big-Data/Cloud [39, 38, 52, 20, 17], and HPC/Grid computing settings [53, 31, 45]. A large body of work [54, 32, 11] leveraged known MapReduce job structure to accurately predict both resource demand and runtimes across different data input sizes. Our architecture allows using any of these techniques, when the application framework is known, while this paper presents a framework-agnostic solution purely based on history. History-based modeling has been used in other contexts: failure-prediction for quality of service (QoS) [18], and resource allocation in business process management [28, 3, 35].

SLO enforcement. Automatic techniques for meeting SLOs [59, 19], use a combination of profiling and job structure knowledge for runtime prediction. PRESS [23] focuses on meeting SLOs at a single-node level and can adjust allocated resources online. Jockey [19] provides a solution for dynamic reprovisioning based on job models derived from execution history and job's internal dependencies. It can be used as a framework-specific dynamic reprovisioning policy. *Morpheus* provides deadlines and global arbitration, which are beyond the scope of [19]. Other dynamic enforcement mechanisms include control-theoretic approaches such as [16, 49].

Online packing and scheduling. The scheduling problem solved by *Morpheus* is a significant generalization of online multidimensional bin-packing problems [4, 25, 7, 5] and online deadline-scheduling problems (see [36, 6] and references therein). Placement in periodic settings has also been studied in the context of real-time and multiprocessor machines [8, 47, 14]. However, the combination of jobs with stage-dependencies, periodicity and and deadlines requires novel algorithm design.

Cluster Scheduling. There has been a substantial body of work on cluster scheduling for big-data analytics [22, 29, 58, 50, 24]. Corral [30] leverages job recurrence and predictable resource requirements to coordinate data and task placement for higher utilization, but does not consider SLOs. Based on published material, SLO inference/enforcement is not present in Mesos [27], Borg [55], and Omega [44]. However, *Morpheus*' mechanisms can be adapted to alternative underlying schedulers. Apollo [9] makes more explicit trade-offs on time vs locality at the task level, but does not provide job-completion SLOs. YARN's reservation system [13] serves as a base for *Morpheus*, but it left unsolved the SLO and job resource model derivation, support for periodic reservations, and dynamic reprovisioning. Moreover, the packing algorithms we present here outperform the one in [13] even for non-periodic jobs [1].

11 Conclusion

In this paper, we present *Morpheus*, a system designed to resolve the tension between predictability and utilization—that we discovered thorough analysis of cluster workloads and operator/user dynamics. *Morpheus* builds on three key ideas: automatically deriving SLOs and job resource models from historical data, relying on recurrent reservations and packing algorithms to enforce SLOs, and dynamic reprovisioning to mitigate inherent execution variance. We validate our design and implementation against large production traces, and on a 2700-node cluster. *Morpheus* reduces worst-case SLO violations by 5-13×, while concurrently reducing the cluster footprint by 14-28%. Overall, *Morpheus* enables predictable performance with less resource provisioning—a win-win for operators and users.

Acknowledgements

We thank our shepherd Sasha Fedorova, and the reviewers for their insightful feedback. We are particularly indebted to Chris Douglas for numerous conversations that helped shape this project. We are also grateful to many colleagues for many great discussions: Ricardo Bianchini, Roni Burd, Kishore Chaliparambil, Chris Douglas, Avrilia Floratou, Giovanni M. Fumarola, Greg Ganger, Brighten Godfrey, Solom Heddaya, Virajith Jalaparti, Alekh Jindal, Srikanth Kandula, Konstantinos Karanasos, Alica Li, Joseph Naor, Vivek Narasayya, Sekhar Pasupuleti, Sean Po, Raghu Ramakrishnan, Keerthi Selvaraj, Arun Suresh, Vin Wang, and Markus Weimer.

References

[1] LowCost: A Cost-Based Placement Agent for YARN Reservations. https://issues.apache.org/jira/browse/YARN-3656.

[2] Support for recurring reservations in the YARN Reservation System. https://issues.apache.org/jira/browse/YARN-5326.

[3] M. Arias, E. Rojas, J. Munoz-Gama, and M. Sepúlveda. A framework for recommending resource allocation based on process mining. In *Business Process Management Workshops - BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers*, pages 458–470, 2015.

[4] J. Augustine, S. Banerjee, and S. Irani. Strip Packing with Precedence Constraints and Strip Packing with Release Times. *Theoretical Computer Science*, 410(38-40), 2009.

[5] Y. Azar, I. R. Cohen, and I. Gamzu. The loss of serving in the dark. In *Proceedings of the Symposium on Theory of Computing Conference*, STOC, 2013.

[6] Y. Azar, I. Kalp-Shaltiel, B. Lucier, I. Menache, J. S. Naor, and J. Yaniv. Truthful online scheduling with commitments. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation*, pages 715–732. ACM, 2015.

[7] N. Bansal and A. Khan. Improved Approximation Algorithm for Two-Dimensional Bin Packing. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, 2014.

[8] S. K. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 280–288, Washington, DC, USA, 1995. IEEE Computer Society.

[9] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. 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, Oct. 2014. USENIX Association.

[10] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007.

[11] L. Cherkasova. Performance modeling in Mapreduce environments: Challenges and opportunities. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 5–6, New York, NY, USA, 2011. ACM.

[12] T. P. P. Council. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html*, 2008.

[13] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC, 2014.

[14] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

[15] A. Desai, K. Rajan, and K. Vaswani. Critical path based performance models for distributed queries. In *Microsoft Tech-Report: MSR-TR-2012-121*, 2012.

[16] Y. Diao, J. L. Hellerstein, S. Member, S. Parekh, S. Member, R. Griffith, G. E. Kaiser, S. Member, and D. Phung. A control theory foundation for self-managing computing systems. *IEEE journal*, 23:2213–2222, 2005.

[17] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Y. Zomaya, and B. B. Zhou. Profiling applications for virtual machine placement in clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 660–667. IEEE, 2011.

[18] J. Ejarque, A. Micsik, R. Sirvent, P. Pallinger, L. Kovacs, and R. M. Badia. Semantic resource allocation with historical data based predictions. In *The First International Conference on Cloud Computing, GRIDs, and Virtualization, CLOUD COMPUTING 2010*, 2010.

[19] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys, 2012.

[20] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 87–92. IEEE, 2010.

[21] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 592–603. IEEE, 2009.

[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[23] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16, Oct 2010.

[24] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2014.

[25] R. Harren and W. Kern. Improved Lower Bound for Online Strip Packing. *Theory of Computing Systems*, 56(1), 2015.

[26] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[27] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[28] Z. Huang, W. Aalst, X. Lu, and H. Duan. Reinforcement Learning Based Resource Allocation in Business Process Management. *Data and Knowledge Engineering*, 70(1):127 –145, 2011.

[29] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[30] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. 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.

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

[32] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM.

[33] K. Lee, A. C. Konig, V. Narasayya, B. Ding, S. Chaudhuri, B. Ellwein, A. Eksarevskiy, M. Kohli, J. Wyant, P. Prakash, R. Nehme, J. Li, and J. Naughton. Operator and query progress estimation in microsoft sql server live query statistics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2016)*. ACM Association for Computing Machinery, June 2016.

[34] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1-3):259–271, 1990.

[35] T. Liu, Y. Cheng, and Z. Ni. Mining event logs to support workflow resource allocation. *Knowl.-Based Syst.*, 35:320–331, 2012.

[36] B. Lucier, I. Menache, J. S. Naor, and J. Yaniv. Efficient online scheduling for deadline-sensitive jobs. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 305–314. ACM, 2013.

[37] R. Mavlyutov, C. Curino, B. Asipov, and P. Cudre-Mauroux. Dependency-Driven Analytics: a Compass for Uncharted Data Oceans, 2016. Microsoft Technical Report MS-TR-2016-69, `http://bit.ly/2dQfRhc`.

[38] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 507–518. ACM, 2010.

[39] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of mapreduce pipelines. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 681–684. IEEE, 2010.

[40] J. Ortiz, V. T. de Almeida, and M. Balazinska. Changing the face of database cloud services with personalized service level agreements. In *CIDR*, 2015.

[41] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. PREDIcT: Towards predicting the runtime of large scale iterative analytics. *Proceedings of the VLDB Endowment*, 6(14):1678–1689, 2013.

[42] A. D. Popescu, V. Ercegovac, A. Balmin, M. Branco, and A. Ailamaki. Same queries, different data: Can we predict runtime performance? In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 275–280. IEEE, 2012.

[43] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 36. ACM, 2016.

[44] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys, 2013.

[45] O. Sonmez, N. Yigitbasi, A. Iosup, and D. 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.

[46] J. Spillner and A. Schill. Dynamic SLA template adjustments based on service property monitoring. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 183–189. IEEE, 2009.

[47] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002.

[48] The Apache Software Foundation. GridMix, 2015. `http://hadoop.apache.org/docs/current/hadoop-gridmix/GridMix.html`.

[49] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[50] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. 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.

[51] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[52] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. 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.

[53] 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.

[54] A. Verma, L. Cherkasova, and R. 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.

[55] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[56] K. Wang and M. M. H. Khan. Performance prediction for apache spark platform. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE*

12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on, pages 166–173. IEEE, 2015.

[57] P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour. *Service level agreements for cloud computing*. Springer Science & Business Media, 2011.

[58] 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 *Proceedings of the ACM European Conference on Computer Systems*, EuroSys, 2010.

[59] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 53–62, New York, NY, USA, 2012. ACM.